Aleksi Koskinen

# Android-sovelluksen toteutus OpenGL ES –kirjaston ja Native Development Kitin avulla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

03.05.2018

Metropolia

| | |
|---|---|
| Tekijä<br>Otsikko<br><br>Sivumäärä<br>Aika | Aleksi Koskinen<br>Android-sovelluksen toteutus OpenGL ES –kirjaston ja Native Development Kitin avulla<br><br>59 sivua<br>03.05.2018 |
| Tutkinto | Insinööri (AMK) |
| Tutkinto-ohjelma | Tietotekniikka |
| Ammatillinen pääaine | Ohjelmistotekniikka |
| Ohjaajat | Lehtori Juha-Pekka Kämäri |

Android on viime vuosina ottanut hallitsevan käyttöjärjestelmän roolin älypuhelimissa. Tässä insinöörityössä tarkastellaan erityisesti sitä, miten Androidiin voi tehdä graafisia sovelluksia eri tekniikoilla.

Työ aloitetaan kertomalla historiaa Android-käyttöjärjestelmästä. Sen arkkitehtuuria katsotaan tarkemmin ja sitä, mitä sen eri kerroksissa tapahtuu. Lisäksi katsotaan, miten Android on kehittynyt vuosien varrella aina tämän päivän Androidiin asti.

Pohjustuksen jälkeen tutustutaan OpenGL-kirjaston Androidille tehtyyn haaraan nimeltä OpenGL ES. Työssä lähdetään ensin katsomaan, miten graafisia sovelluksia voi tehdä käyttäen OpenGL ES -versiota 1.0. Katsotaan, miten siinä voi piirtää puhelimen näytölle 2D-muotoja kuten kolmion. Katsotaan myös, miten piirrettyyn kolmioon voi lisätä värejä ja tekstuureja. Tarkastellaan myös, miten projektioita ja matriiseja käytetään, jotta piirretyt muodot saadaan näkymään käyttäjälle halutulla tavalla. Osion lopussa katsotaan, miten OpenGL ES:n uudet versiot kuten 2.0 ja 3.0 eroavat versiosta 1.0. Tutustutaan tarkemmin niiden oleellisimpaan eroon eli version 2.0 tuomien varjostimien käyttöön.

OpenGL ES -osion jälkeen työssä tarkastellaan Native Development Kitiä (NDK). Katsotaan sen tarjoamia eri työkaluja, jotka mahdollistavat natiivien sovelluksien tekemisen Androidiin käyttäen C/C++-koodia. Hyödynnetään näitä työkaluja, ja tarkastellaan miten natiivin koodin avulla voi käyttää OpenGL ES -kirjastoa ja piirtää muotoja. Osion lopussa katsotaan myös hieman Vulkania, jonka on sanottu olevan OpenGL ES:n korvaaja.

Työn lopussa tehdään vielä hyvin yksinkertainen 3D-peli käyttäen OpenGL ES -kirjaston versiota 3.0. Peliä tarkastellessa katsotaan samalla, miten Androidille voi tehdä 2D-sovelluksien lisäksi 3D-sovelluksia.

Työn tuloksena ollaan todettu miten uusimpia OpenGL ES –kirjastoja käytetään 3D-grafiikkaa piirrettäessä. Lisäksi ollaan todettu miten NDK:ta käytetään ja katsottu miksi sitä ylipäätänsä kannattaa käyttää.

| | |
|---|---|
| Avainsanat | Android, OpenGL ES, NDK, Vulkan |

| | |
|---|---|
| Author<br>Title<br><br>Number of Pages<br>Date | Aleksi Koskinen<br>Implementing an Android application using the OpenGL ES library and Native Development Kit<br><br>59 pages<br>03.05.2018 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information technology |
| Professional Major | Software engineering |
| Instructors | Lecturer Juha-Pekka Kämäri |

In recent years, Android has taken over the dominant operating system role in smartphones. This thesis focuses on how Android can perform graphical applications with different technologies.

Work begins by telling a little about the history of the Android operating system. It looks a little bit more about its architecture and what happens on its different layers. In addition, the work takes a look on how Android has developed over the years to today's Android.

Afterwards, you will be introduced to the OpenGL library's Android-based branch named OpenGL ES. In this work, we begin by looking at how graphical applications can be made using OpenGL ES version 1.0. We will see how it can draw 2D forms, such as a triangle, on the phone screen. We will also see how we can add colors and textures to the drawn triangle. The work also considers how projections and matrices are used to make the forms visible to the user as desired. The end of the section looks at how new versions of OpenGL ES such as 2.0 and 3.0 are different from version 1.0. Learn more about their most significant difference, ie use of shaders that came with version 2.0.

After the OpenGL ES section, the work examines the Native Development Kit (NDK). It takes a look at the various tools it offers to enable native applications to Android using the C/C++ code. We utilize these tools and look at how native code can use the OpenGL ES library and draw shapes. At the end of the section, the work takes a little look about Vulkan, which is said to be the replacement for OpenGL ES.

At the end of the work, a very simple 3D game will be made using the OpenGL ES library version 3.0. As you look at the game, you can see how Android can do 3D applications in addition to 2D apps.

As a result of this work, we have found how the latest OpenGL ES libraries are used to draw 3D graphics. In addition, we have learned how to use the NDK and see why it is worth using it at all.

| | |
|---|---|
| Keywords | Android, OpenGL ES, NDK, Vulkan |

## Sisällys

## Lyhenteet

| | |
|---|---|
| APK | Android-käyttöjärjestelmän käyttämä tiedostomuoto pakkauksille, joita Android käyttää mobiilisovellusten jakeluun ja asennukseen. |
| JVM | Java Virtual Machine. Abstrakti laskentakone, jonka avulla tietokone voi suorittaa Java-ohjelman. |
| DVM | Dalvik Virtual Machine. Android-käyttöjärjestelmän virtuaalikone, joka vastaa joidenkin Androidin järjestelmäkirjastojen sekä kaikkien Google Play -sovellukkaupan sovellusten suorittamisesta. |
| ART | Android Runtime. Androidin käyttämä sovelluskehitysympäristö, joka korvasi Dalvikin. |
| API | Application Programming Interface. Ohjelmointirajapinta, jonka mukaan eri ohjelmat voivat tehdä pyyntöjä keskenään. |
| GPU | Graphics Processing Unit. Tietokoneen näytönohjain, joka piirtää grafiikan tietokoneen näytölle. |
| NDK | Native Development Kit. Työkalu, joka mahdollistaa Android-sovelluksien tekemisen C/C++-kielillä. |
| JNI | Java Native Interface. Ohjelmointikehys, joka mahdollistaa Javan ja natiivin koodin yhteistyön. |
| GLSL | OpenGL Shading Language. Korkeantason varjostinkieli, jossa on C-ohjelmointikielen mukainen syntaksi. |
| SPIR | Standard Portable Intermediate Representation. Keskitason kieli rinnakkaiseen laskentaan ja grafiikkaan. |

# 1 Johdanto

Tämän työn tarkoituksena on perehtyä Android-käyttöjärjestelmään ja tarkastella sitä varsinkin graafisen toteutuksen näkökulmasta. Android on tällä hetkellä maailman suosituin mobiilikäyttöjärjestelmä, ja varsinkin siihen tehdyt pelit ovat erittäin suosittuja. Tässä työssä katsotaan, millä eri tekniikoilla Androidiin saatiin ohjelmoitua ihan niitä ensimmäisiä graafisia sovelluksia kuten pelejä ja millä tekniikoilla nykypäivän graafiset sovellukset tehdään.

Työ on karkeasti jaettu kolmeen osaan. Ensimmäisessä osiossa tarkastellaan, miten OpenGL ES -kirjastoa voidaan hyödyntää graafisten sovelluksien tekemiseen. Katsotaan, miten kyseisen kirjaston ensimmäisellä versiolla 1.0 saatiin graafisia sovelluksia aikaiseksi sekä tarkastellaan, miten OpenGL ES -kirjasto ja sen tekniikat on kehittynyt vuosien varrella aina versioon 3.2 asti.

Toisessa osiossa tarkastellaan, miten Androidiin voi tehdä graafisia sovelluksia käyttäen Native Development Kitiä. Tämä tarkoittaa, että täytyy siirtyä Javan mukavuusalueelta pois, ja kehittää sovelluksia käyttäen natiivia koodia ohjelmointikielien C/C++ avulla. Tutkitaan, miten natiivi koodi ja Java pystyy tekemään yhteistyötä JNI:n avulla. Lisäksi tässä osiossa tutkitaan hieman uuden sukupolven grafiikkakirjastoa, Vulkania. Tutkitaan, mitä eroa Vulkanissa on OpenGL ES:ään ja mitä kehittäjältä vaaditaan Vulkan-sovelluksen tekemiseen.

Viimeisessä osiossa käydään läpi itse työtä varten tehtyä yksinkertaista 3D-peliä. Käytetään pelin tekemiseen OpenGL ES -versiota 3.0, jolloin voidaan pelin läpikäynnin yhteydessä tarkastella sitä, miten saadaan 3D-peli aikaiseksi uusimmilla OpenGL ES -versioilla.

# 2 Android-käyttöjärjestelmä

## 2.1 Historia

Android on tällä hetkellä maailman eniten asennettu käyttöjärjestelmä mobiilialustoihin. Sen kehitys on lähtenyt käyntiin vuonna 2003, kun Yhdysvalloissa perustettiin Android

Inc. -niminen yritys. Yksi sen perustajista, Andy Rubin kertoi, että heidän tarkoituksenaan oli tehdä älykkäämpiä mobiililaitteita, jotka olisivat tietoisempia omistajan sijainnista ja mieltymyksistä [1].

Vuonna 2005 alkoi spekulointi siitä, haluaako Google ryhtyä kilpailemaan matkapuhelinalalla, kun se päätti ostaa Android Inc:in vuoden 2005 elokuussa [2]. Google palkkasi Android Inc:in jäsenet jatkamaan työskentelyään Androidin parissa, joista esimerkiksi Andy Rubin alkoi työskentelemään Googlelle Android-kehityksen johtajana.

Tärkeä osa Androidin tulevaa menestystä oli vuonna 2007 perustettu Open Handset Alliance (OHA). Siihen kuuluu 84 laitteisto- ja ohjelmistovalmistajaa sekä teleoperaattoriyritystä, mukaan lukien HTC, Qualcomm, Motorola ja Samsung [3]. Sen tarkoituksena on kehittää avoimeen lähdekoodiin perustuvia standardeja mobiilialustoille. Sen yksi perustajajäsen ja suurin tukija on Google, jonka avulle he vuonna 2007 julkaisivatkin heidän ensimmäisen yhteisen tuotoksensa, Androidin. Vuoden 2008 syyskuussa tuli ensimmäinen kaupallinen Android-älypuhelin markkinoille, HTC Dream. Koska Google on Open Handset Alliancen yksi perustajajäsenistä, niin Android vahvasti pohjautuu Googlen palveluihin.

Ei mennyt kauan, kunnes Android saavutti älypuhelimien markkinajohtaja-aseman. Vuoden 2010 viimeisellä neljänneksellä Googlen Android oli saavuttanut 33,3 % markkinaosuuden, ohittaen Nokian, joka oli toisena 31 %:n osuudella. Vuoden 2017 ensimmäisellä neljänneksellä Android oli jo saavuttanut 83,4 % markkinaosuuden, kun toisena tuli iOS Applelta 15,4 % osuudella. Kolmantena olivat Windows-puhelimet, joilla oli vain 0,1 % markkinaosuus. [3; 4.] Tämä kertoo siitä, että Android ja iOS hallitsevat älypuhelinmarkkinoita melkein täydellisesti.

## 2.2 Rakenne

Android on avoimeen lähdekoodiin perustuva käyttöjärjestelmä, joka mahdollistaa ohjelmistokehittäjiä muokkaamaan suurinta osaa sen koodista. Androidin käyttöjärjestelmä on rakennettu Linux Kernel -ytimen ympärille. Alun perin se perustui Kernel-versioon 2.6, mutta sitäkin on muiden päivitysten ohessa päivitetty, joten vuonna 2017 suurin osa Android-laitteista perustui versioihin 3.18 ja 4.4 Linux Kernelistä. [2.]

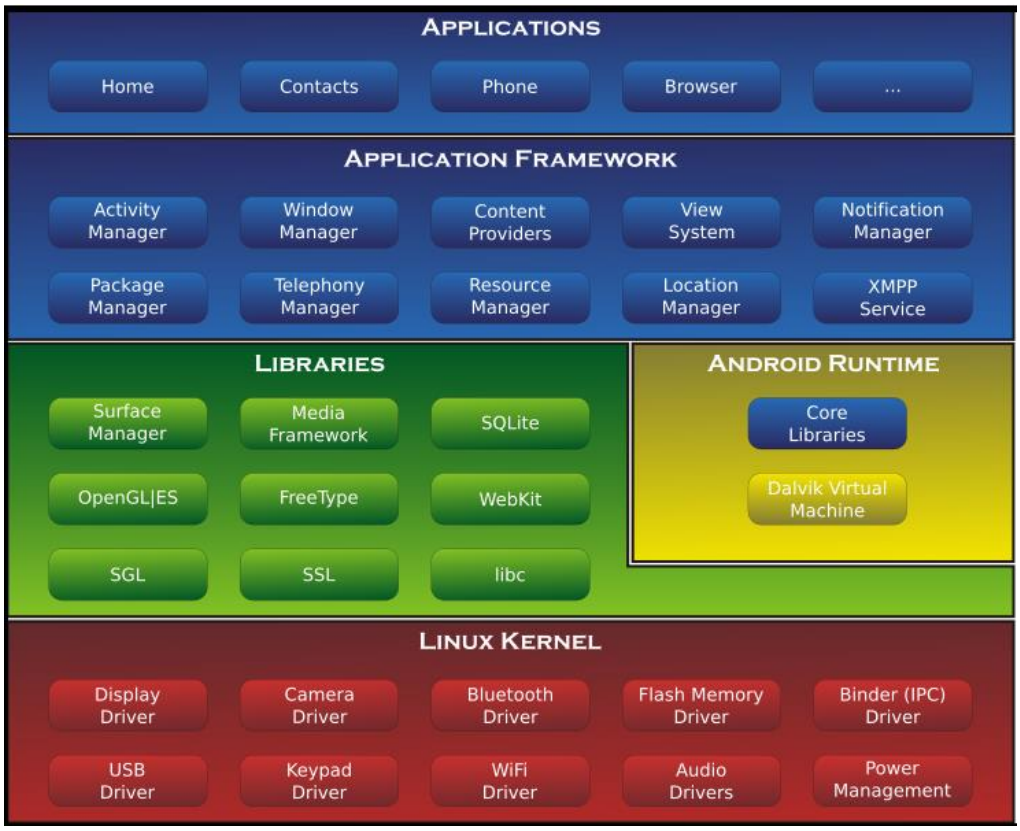
Vaikkakin Kernel-tason koodi on kirjoitettu C-ohjelmointikielellä, niin Android-käyttöjärjestelmän pääohjelmointikieli on Java. Suurin osa sen kirjastoista on kirjoitettu Javalla. Android ei kuitenkaan käytä Javan omaa virtuaalikonetta (JVM), vaan siihen on rakennettu aivan oma virtuaalikone nimeltä Dalvik (Uusimmissa Android-versioissa korvattu Android Runtimella). Koska Java on Androidin pääohjelmointikieli, niin sovellukset ovat kirjoittamisen jälkeen .java-muodossa. Kun se halutaan ajaa, niin ensin .java-tiedosto menee Java-kääntäjään, joka tekee siitä .class-tiedoston. Tämän jälkeen se viedään Androidin omaan dx-ohjelmaan, joka kääntää .class-tiedoston .dex-muotoon. Generoitunut .dex-tiedosto sitten yhdessä mahdollisten kuva- ja .xml-tiedostojen kanssa annetaan Dalvik-virtuaalikoneelle, joka pakkaa ne yhdeksi .apk-tiedostoksi. [6.] Tämän käyttäjä voi ajaa ja asentaa sovelluksen omaan Android-laitteeseensa.

Syksyllä 2013, Google julkaisi Android 4.4 (KitKat)-julkaisun yhteydessä Android Runtimen (ART). Se on uudenlainen sovelluskehitysympäristo, jonka olisi tarkoitus korvata Dalvik kokonaan. Android Runtime käyttää käyttää ahead-of-time (AOT)-kääntäjää Dalvikin just-in-time (JIT)-kääntäjän sijasta. Se kääntää sovelluksen valmiiksi jo sen asennuksen yhteydessä. Poistamalla Dalvikin tulkinta- ja JIT-kokoelman, Android Runtime parantaa yleistä suorituskyvyn tehokkuutta ja vähentää virrankulutusta, mikä parantaa akun autonomiaa mobiililaitteissa. Samanaikaisesti ART tuo käyttöön sovellusten nopeamman toteutuksen, parannetun muistin allokoinnin ja jätteidenkeräysjärjestelmän (GC) mekanismit, uusien sovellusten virheenkorjausominaisuudet sekä tarkemmat sovellusten korkean tason profiloinnin. [7.] Vaikka Android 4.4 toi käyttäjille mahdollisuuden käyttää Android Runtimeä, niin siinä Dalvik oli vielä oletus-virtuaalikone. Vasta 2014 loppupuolella Android 5 (Lollipop)-julkaisussa ART korvasi kokonaan Dalvikin.

Jotta ymmärtäisimme paremmin, miten Android toimii, niin katsotaan kuvaa 1, joka näyttää Androidin eri käyttöjärjestelmäkerroksia. Kuvasta nähdään, että Android on jaettu 5 eri osioon, neljässä eri kerroksessa:

- Linux Kernel - Tämä on se kerneli, mihin Android perustuu. Se sisältää kaikki alemman tason laitteiden ajurit.
- Libraries - Nämä sisältävät koodin, joka tarjoaa Android-käyttöjärjestelmän pääominaisuudet. Esimerkiksi SQLite-kirjasto tarjoaa tietokantatukea niin, että sovellukset voivat käyttää sitä datan tallennukseen. WebKit-kirjasto taas tarjoaa toimintoja Internetin selaamiseen. Tässä kerroksessa on myös OpenGL|ES-kirjasto, jota tässä työssä tarkastellaan tarkemmin myöhemmin.

- Android Runtime – Tämä sijaitsee samassa kerroksessa Libraries-kerroksen kanssa, ja se tarjoaa joukon ydinkirjastoja, joiden avulla kehittäjät voivat kirjoittaa Android-sovelluksia. Android Runtime sisältää myös Dalvik-virtuaalikoneen, joka mahdollistaa jokaisen Android-sovelluksen pyörimisen omassa prosessisa, jolla on oma ilmentymä Dalvik-virtuaalikoneesta.
- Application Framework – Application Framework -kerros paljastaa Android-käyttöjärjestelmän eri ominaisuudet sovelluskehittäjille, jotta he voivat käyttää niitä sovelluksissaan.
- Applications – Tällä ylimmäisellä kerroksella ovat käyttöjärjestelmän valmiit sovellukset, sekä kaikki ne sovellukset, jotka käyttäjä on itse ladannut ja asentanut Android Play Storesta. Myös kaikki kehittäjien itse kirjoittamat sovellukset sijaitsevat tässä kerroksessa.




Kuva 1. Android-käyttöjärjestelmän arkkitehtuuri [2].

## 2.3 Versiot

Android-käyttöjärjestelmän versiohistoria alkoi vuoden 2008 syyskuussa, kun ensimmäinen kaupallinen versio Android 1.0 julkaistiin. Siitä lähtien Androidia kehitetään

jatkuvasti Googlen ja Open Handset Alliancen toimesta, minkä vuoksi uusia versioita julkaistaan tasaisin väliajoin. [8.] Kuten taulukosta 1 näkyy, kahdelle ensimmäiselle versiolle ei vielä annettu tiettyä nimeä, mutta versiosta 1.5 (Cupcake) lähtien jokaiselle uudelle versiolle annettiin uusi nimi, joka on nimetty aakkosjärjestyksessä jonkun makean ruuan nimen mukaan.

Taulukko 1. Androidin eri versioiden nimet ja API-tasot [8].

| Code name | Version number | Initial release date | API level | Security patches[1] |
|---|---|---|---|---|
| (No codename)[2] | 1.0 | September 23, 2008 | 1 | Unsupported |
| (Internally known as "Petit Four")[2] | 1.1 | February 9, 2009 | 2 | Unsupported |
| Cupcake | 1.5 | April 27, 2009 | 3 | Unsupported |
| Donut[3] | 1.6 | September 15, 2009 | 4 | Unsupported |
| Eclair[4] | 2.0 – 2.1 | October 26, 2009 | 5 – 7 | Unsupported |
| Froyo[5] | 2.2 – 2.2.3 | May 20, 2010 | 8 | Unsupported |
| Gingerbread[6] | 2.3 – 2.3.7 | December 6, 2010 | 9 – 10 | Unsupported |
| Honeycomb[7] | 3.0 – 3.2.6 | February 22, 2011 | 11 – 13 | Unsupported |
| Ice Cream Sandwich[8] | 4.0 – 4.0.4 | October 18, 2011 | 14 – 15 | Unsupported |
| Jelly Bean[9] | 4.1 – 4.3.1 | July 9, 2012 | 16 – 18 | Unsupported |
| KitKat[10] | 4.4 – 4.4.4 | October 31, 2013 | 19 – 20 | Unsupported[11] |
| Lollipop[12] | 5.0 – 5.1.1 | November 12, 2014 | 21 – 22 | 5.1.x Supported |
| Marshmallow[13] | 6.0 – 6.0.1 | October 5, 2015 | 23 | Supported |
| Nougat[14] | 7.0 – 7.1.2 | August 22, 2016 | 24 – 25 | Supported |
| Oreo[15] | **8.0 – 8.1** | August 21, 2017 | 26 – 27 | Supported |
| **Legend:** Old version | Older version, still supported | **Latest version** | | |

Huomioitavaa on myös se, että jokaisen uuden Android-version yhteydessä julkaistaan sen uusien ominaisuuksien lisäksi myös uusi API-taso. Jos kehittäjä tekee uuden sovelluksen, joka on kirjoitettu Androidin uusimmalle versiolle ja API-tasolle, niin vain käyttöliittymät, jotka käyttävät samaa API-tasoa, pystyvät sovelluksen käynnistämään. Siksi on heti sovelluskehityksen alkuvaiheissa tärkeää määrittää sovellukselle sopiva minimiversio, jotta myös käyttöliittymät, joissa on vanhempi Android-versio, pystyvät suorittamaan sovelluksen.

Androidin versiot ovat hyvin jakautuneita käyttäjien kesken. Vuoden 2017 joulukuussa todettiin, että yksikään Android-versio ei ollut saanut yli 30 prosentin suosiota kaikista Android-versioista. Suurimmissa osissa Android-laitteita on Marshmallow (29,7%), ja

toisiksi eniten laitteissa on Lollipop (26,3%). Nougat oli saanut jo kunnioitettavan (23,3%) osuuden, vaikka se on vasta vähän yli vuoden vanha versio. Uusimman version, Oreon, osuus laitteista on vielä alhainen (0,5%). [9] Tämä johtunee siitä, että versio on vain vähän aikaan sitten julkaistu, ja se pyörii vain pääosin laitteissa, jotka Google on tehnyt.

# 3 Graafinen toteutus OpenGL ES -kirjaston avulla

Android-kehys tarjoaa useita vakiintuneita työkaluja graafisten hyvien ja toimivien käyttöliittymien luomiseen. Jos kehittäjä kuitenkin haluaa paremman käsityksen ja hallinnan siitä, mitä sovellus näyttää ruudulla, tai jos kehittäjä haluaa tehdä kolmiulotteista grafiikkaa, tarvitsee hän tiettyjä työkaluja siihen. Android-kehys tarjoaa kehittäjille OpenGL ES API -kirjaston, joka tarjoaa joukon työkaluja, joilla kehittäjät voi tehdä korkealaatuisia, ja aivan niin monimutkaisia 2D- ja 3D- sovelluksia kuin he itse haluavat. Itse OpenGL (Open Graphics Library) on hyvin laaja graafinen 2D- ja 3D-rajapinta, jota voi käyttää melkein missä tahansa alustassa. Android käyttää tästä kirjastosta sulautetuille järjestelmille tarkoitettua ES-haaraa. OpenGL ES onkin maailman laajin käytössä oleva 3D- grafiikka API.

OpenGL ES on API-kirjasto, jonka Khronos Group on kehittänyt. Tammikuussa 2000 perustettu Khronos Group on jäsenrahoitettu toimialakonsortio, joka keskittyy avoimien standardien ja rojaltimattomien API-sovellusten luomiseen kannettaville ja sulautetuille laitteille. Khronos Groupiin kuuluu noin 120 yritystä, joista mainittakoon AMD, Apple Inc., Google, Intel, Nvidia, ja Huawei. [10.]

Vuosien aikana OpenGL ES API -kirjastosta on tullut monta eri versiota. Ensimmäinen versio julkaistiin vuonna 2003, ja se oli nimeltään OpenGL ES 1.0. Tästä myöhemmin julkaistiin versiot 1.1, 2.0, 3.0, 3.1 ja viimeisin versio, vuonna 2015 julkisesti julkaistu OpenGL ES 3.2 -versio. Kaikki versiot tarjoavat korkean suorituskyvyn graafisten 2D- ja 3D-sovellusten tekemiseen. Ohjelmointi OpenGL ES 2.0:lle ja 3.0:lle on hyvin pitkälti samanlaista. 3.0-versio tarjoaa vain hieman lisäominaisuuksia 2.0-versioon nähden. Tämän vuoksi jos sovellus on tehty versioon 3.0, ja käyttäjän käyttöliittymä ei tue 3.0 versiota, niin se pystyy silti suorittamaan sovelluksen käyttäen 2.0-versiota ES API:sta. Taasen ohjelmointi 1.0/1.1 (voidaan käyttää nimetystä 1.x) versioille eroaa huomattavasti siitä, mitä se on versioille 2.0/3.0. Jos sovellus on tehty käyttäen 2.0 tai

suurempaa versiota, niin se ei tule pyörimään kohdelaitteessa, joka tukee vain 1.x-versioita. Tästä syystä kehittäjän on tarkkaan mietittävä, mitä versiota tulee käyttämään.

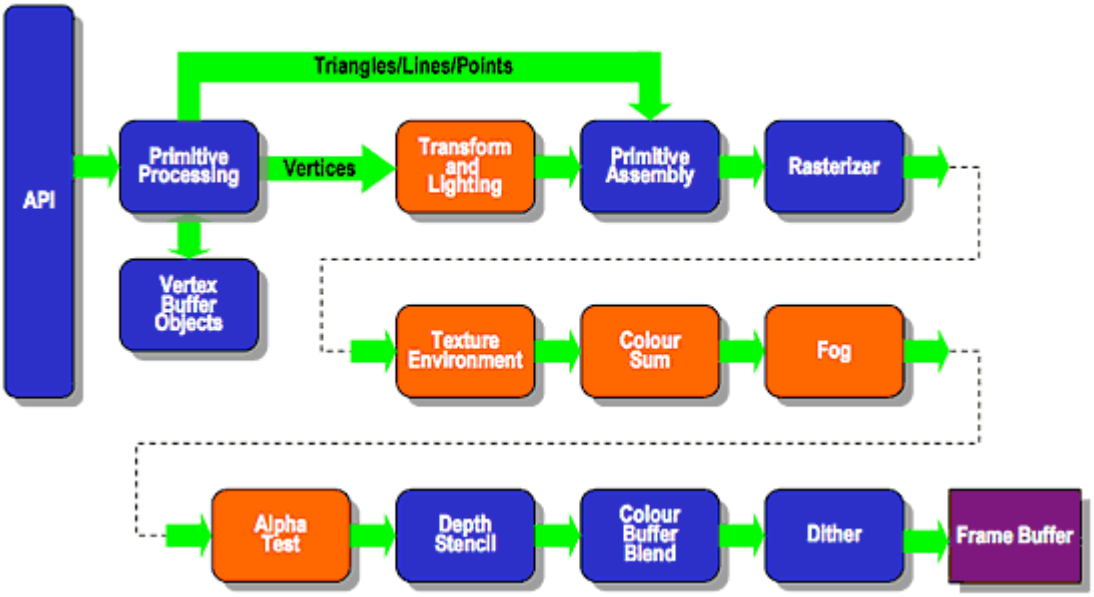
Android tukee OpenGL:tä sekä sen oman API:n kautta (ES), mutta myös Native Development Kitin (NDK) kautta. Tässä työssä perehdytään ensin vähän tarkemmin kyseiseen OpenGL ES API:iin, sekä sen jälkeen katsotaan, miten sovellus kehitetään Native Development Kitin avulla.

## 3.1 OpenGL ES 1.0 / 1.1

Kun lähtee toteuttamaan graafista, tai mitä tahansa muuta Android-sovellusta, niin kehittäjät käyttivät vuoteen 2013 asti yksinomaan Eclipse-ympäristöä sovellusten tekoon. Vuonna 2014 joulukuussa Google julkaisi kuitenkin Android Studio 1.0 -nimisen ohjelman, joka on juuri Android-sovelluksia varten kehitetty ohjelma. Sen oli tarkoitus korvata Eclipse kehitystyössä. Uusin versio on 3.0, joka julkaistiin 2017 lokakuussa. Kaikki työn tulevat esimerkit on toteutettu nimenomaan Android Studio -ohjelmalla.

OpenGL ES 1.x on kaikista versioista se yksinkertaisin ja helposti ymmärrettävin. Tämä johtunee siitä, että OpenGL ES 1.x API ei tarjoa kehittäjille täydellistä pääsyä taustalla oleville laitteille. Useimmat API:n rasterointitoiminnot ovat kovakoodattuja API:iin, josta johtuukin se, että sitä kutsutaan mm. nimellä kiinteän funktion graafinen rasterointi API (fixed-function pipeline). OpenGL ES 2.0 API taasen julkaistiin ohjelmoitavana graafisen esitystapahtuma API:na (programmable pipeline). [11.] Tämä antoi kehittäjille täyden pääsyn taustalla oleviin laitteisiin niin sanottujen varjostimien avulle (shaders).

Kuvasta 2 nähdään OpenGL ES 1.x kiinteiden funktioiden putki, ja kaikki vaiheet, mitä vaaditaan siihen, että lopullinen kuva tulee pikseleinä näytölle. Kiinteän funktion kautta tuotettu grafiikka sisältää laitekohtaisia algoritmeja useimmille rasterointivaikutuksille. Näitä algoritmeja, ja niihin perustuvia rasterointitoimintoja, ei voi muuttaa. Kiinteän toiminnon ansiosta grafiikkalaitteisto voitaisiin optimoida nopeampaan rasterointiin kuin ohjelmoitavassa putkessa. Kiinteiden funktioiden ansiosta OpenGL ES 1.x on kehittäjille huomattavasti yksinkertaisempaa ohjelmoida kuin OpenGL ES 2.0 API. 1.x-versio riittää kuitenkin tuottamaan hyviä graafisia sovelluksia, eli se on ihan hyvä vaihtoehto kehittäjälle, joka haluaa tehdä helposti uuden sovelluksen.




Kuva 2. OpenGL ES 1.x kiinteiden funktioiden putki [12].

- Primitiivinen käsittely (Primitive Processing) – Saa sovellukselta tiedot ja muuntaa ne vertekseiksi, jotka menevät eteenpäin vaiheissa.
- Verteksipuskuriobjektit (Vertex Buffer Objects) – Tietueita, jotka tallentaa verteksit näytönohjaimen muistiin, sallien niiden käytön vain, kun niitä tarvitaan. Objektien piirtäminen nopeutuu, koska verteksit saadaan nopeammin.
- Muunnos ja valaistus (Transform and Lighting) – Muuttaa sille saapuvat verteksit silmäkoordinaatistoon. Kaikille vertekseille lasketaan oma valaistus, ja kaikki verteksit hylätään, jotka eivät ole näkökentässä.
- Primitiivinen kokoaminen (Primitive Assembly) – Koostaa saamiensa verteksien tiedoista geometrisia muotoja, kuten kolmioita, pisteitä, viivoa yms. Kun kaikki geometriset muodot ovat saatu vertekseistä tehtyä, lähetetään muoto seuraavalle vaiheelle.
- Rasterointi (Rasterizer) – Muuttaa saamansa geometrian fragmenteiksi. Fragmentti on joukko tietoa, mitä käytetään yhden pikselin piirtämiseksi kuvapuskuriin.
- Tekstuuriympäristö (Texture Environment) – Jos käytetään tekstuureja, lisätään ne fragmentteihin.
- Värisumma (Colour Sum) – Yhdistää fragmentin mahdollisen tekstuurin, verteksin ja valaistuksen värin fragmentinväriksi.
- Sumu (Fog) – Värjää kaukaisempia fragmentteja saaden aikaan sumu-vaikutelman.
- Läpinäkyvyyden testi (Aplha Test) – Hylkää fragmenttien aplha arvon perusteella liian läpinäkyvät fragmentit.

- Syvyys sapluuna (Depth Stencil) – Katsoo fragmenttien syvyys arvoa, ja katsoo mitkä fragmentit ovat lähimpänä kuvapuskurissa.
- Väripuskurin sekoitus (Colour Buffer Blend) – Laittaa saapuvan fragmentin väripuskuriin.
- Epäröinti (Dither) – Fragmentin sijainnin perusteella vaihe päättää, mikä väri valitaan. Jos väriavo on puolivälissä kahta edustavaa väriä, niin puolet pikseleistä on yksi väri ja puolet toinen.
- Kuvapuskuri (Frame Buffer) – Puskurien joukko, jota voidaan käyttää rasteroinnin kohteena. Sinne säilötään aikaisempien vaiheiden tuottama muoto.

Kun lähtee toteuttamaan graafista sovellusta, joka käyttää kirjastoa, jota eivät välttämättä kaikki Android-laitteet tue, pitää AndroidManifest.xml-tiedostoon määritellä tuettu OpenGL ES -versio esimerkkikoodin 1 tavalla.

```
<uses-feature android:glEsVersion=”0x00010000” android:required=”true” />
```

Esimerkkikoodi 1. Määritetty tuettu OpenGL ES Versio (1.0).

Koska melkein kaikki laitteet tosin tukevat 1.0-versiota, niin sitä tehdessä ei kyseistä määritystä ole tarpeen tehdä. Jos käyttää taasen 2.0- tai 3.1- versiota (määriteltäisiin 0x00030001), niin määritys on hyvä tehdä. Tällöin Google Play -kauppa osaa estää sovelluksen asentamisen laitteeseen, joka ei tue määritettyä versiota. [13]

Kun sovellusta lähtee tekemään, tarvitsee ensinnäkin jonkinlaisen näkymä, joka antaa mahdollisuuden piirtää OpenGL ES -grafiikkaa. API:sta onneksi löytyy tämänlainen näkymä, ja se on nimeltään GLSurfaceView. Näkymä ei itsessään tee paljoakaan, vaan itse objektien piirto kontrolloituu GLSurfaceView.Renderer-rajapinnassa. Kyseinen rajapinta ohjaa sitä, mitä näkymään, johon se on määritetty, piirretään. Renderer-rajapintaan on määritetty esimerkkikoodin 2 mukaisesti kolme metodia, joita Android-järjestelmä käyttää selvittääkseen, mitä ja miten piirtää GLSurfaceView-näkymän.

```
interface Renderer{

      public void onSurfaceCreated(GL10 gl, EGLConfig config);

      public void onSurfaceChanged(GL10 gl, int width, int height);

      public void onDrawFrame(GL10 gl);
}
```

Esimerkkikoodi 2. Rajapinta, joka ohjaa GLSurfaceView-näkymään tulevan piirron.

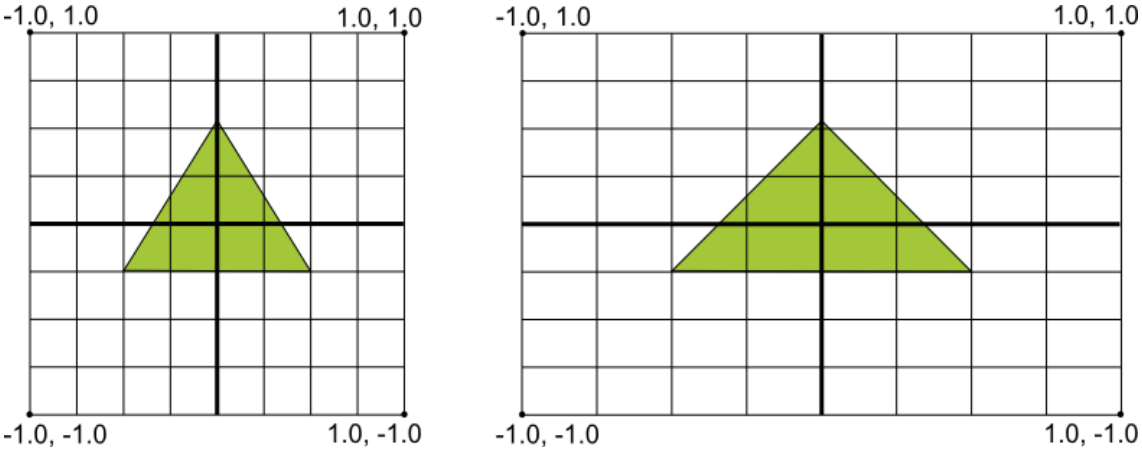
onSurfaceCreated()-metodia kutsutaan joka kerta, kun GLSurfaceView-pinta luodaan. Tämä tapahtuu, kun ensimmäisen kerran käynnistämme toiminnan (aktiviteetin), ja joka kerta, kun palaamme näkymään keskeytetystä tilasta. Tämä tarkoittaa sitä ikävää puolta, että aina, kun aktiviteetti on keskeytystilassa, niin GLSurfaceViewin näkymäpinta tuhoutuu. Kun aktiviteettiin palataan, niin pinta luodaan uudestaan tällä metodilla. Tämä ei muuten olisi niin hirveän huono asia, mutta kaikki OpenGL ES -tilat ja tekstuurit, mitä tähän mennessä kehittäjä on asettanut, menetetään. Tätä ongelmaa kutsutaan kontekstihäviöksi. [14, s. 279.] Kyseisen ongelman takia sovelluksen suunnittelussa saa olla tarkkana, jotta se asianmukaisesti käsittelee tätä kontekstikatoa. Metodissa on myös kaksi parametria. GL10-instanssi avulla kehittäjä voi antaa komentoja OpenGL ES: lle, ja EGLConfig kertoo vain pinnan ominaisuuksista kuten väristä ja syvyydestä. Sen voi yleensä sivuuttaa.

onSurfaceChanged()-metodia kutsutaan aina, kun pinnan kokoa muutetaan. Eli kun käyttäjä vaikka laittaa puhelimen vaakatasoon, niin silloin pinta muuttuu ja kyseistä metodia kutsutaan. Parametreina saadaan uusi leveys ja korkeus, jotka voi sitten GL10-instanssille kertoa, jotta saadaan tehtyä sopivat muutokset.

onDrawFrame()-metodia kutsutaan aina, kun näkymään piirretään jotain uutta. Metodi siis suorittaa kaiken rasteroinnin.

### 3.1.1 Muotojen piirtäminen

Tyypillisin muoto, mitä voi piirtää, on kolmio. Jos haluaa piirtää jotain vähän monimutkaisempaa, niin voi piirtää neliön, joka saadaan laittamalla kaksi kolmiota vastakkain. Kun lähtee piirtämään kolmiota, pitää sovellukselle kertoa hieman kolmiosta, mitä lähdetään piirtämään. Kolmiolla on 3 pistettä, ja jokaista pistettä kutsutaan nimellä verteksi. Verteksillä on paikka 3D-avaruudessa, joka määritetään x- (leveys), y- (korkeus) ja z- (syvyys) koordinaatistossa.




Kuva 3. Kolmio piirrettynä OpenGL ES -koordinaattisysteemiin. Vasemmalle on kolmio oletuskoossa, ja oikealla kolmio skaalattuna Android-laitteen näyttöön. [13]

Kuvaan 3 on oikealle piirretty skaalattu kolmio Android-laitteen näytölle. Tämä on toteutettu käyttämällä projektiotiloja ja kameranäkymiä koordinaattien muokkaamiseen, jotta piirrettävillä objekteilla on oikeat mittasuhteet millä tahansa näytöllä. Esimerkkikoodin 3 mukaan on määritetty kolmion 3 eri pisteverteksiä.

```
public class Triangle {

	private final FloatBuffer vertexBuffer;

	//verteksien koordinaattien määrä
	static final int coords_per_vertex = 3;
	static float triangleCoords[] = {
		0.0f,  0.75f, 0.0f,// Ylin piste
		-0.5f, -0.3f, 0.0f,// Vasen ala piste
		0.5f, -0.3f, 0.0f // Vasen oikea piste
	};

	public Triangle() {
		// Tehdään bytebuffer muistin tallennusta varten
		ByteBuffer bb = ByteBuffer.allocateDirect(
				triangleCoords.length * 4);

		bb.order(ByteOrder.nativeOrder());

		// Laitetaan bufferi käytämään float arvoja
		vertexBuffer = bb.asFloatBuffer();
		// lisätään koordinaatit bufferiin
		vertexBuffer.put(triangleCoords);
		// aloitetaan koordinaattien luku ensimmäisestä koordinaatista
		vertexBuffer.position(0);
	}
}
```

Esimerkkikoodi 3. Kolmion pisteiden koordinaatit.

Esimerkkikoodin 3 ensimmäisen pisteen koordinaateissa on 3 arvoa. 0.0f kertoo pisteen sijainnin koordinaatiston vaakatasossa, eli se on keskellä kohdenäyttöä. 0.75f kertoo sijainnin pystytasossa, eli se on kohdenäytön yläreunan ja keskikohdan puolessavälissä. 0.0f kertoisi syvyyden, mutta tässä sen voi laittaa 0.0f. Huomataan, että jokaisella verteksillä on siis laitettu 3 koordinaattia, joten coords_per_vertex on tällöin kolme.

Koska OpenGL ES on oikeastaan C-kieleen perustuva, ei siinä voi suoraan käyttää javan taulukoita. Sen sijaan pitää käyttää Javan buffereita, joihin voidaan tallentaa dataa. [14, s. 293] Esimerkkikoodissa 3 nähdään, kuinka ByteBuffer tehdään sen allocateDirect()-metodilla. Metodiin annetaan parametreiksi se muistin määrä, jota tarvitaan. Koska kyseisessä esimerkissä käytetään float-arvoja (yksi float-arvo vie 4 tavua muistia), ja annettuja koordinaatteja oli 9, niin muistia täytyy varata 4 x 9 eli 36 tavua. Lopuksi vielä laitetaan koordinaattien positio 0, eli aloitetaan niiden luku ensimmäisestä koordinaatista.

Itse kolmion piirto onnistuu esimerkkikoodin 4 avulla.

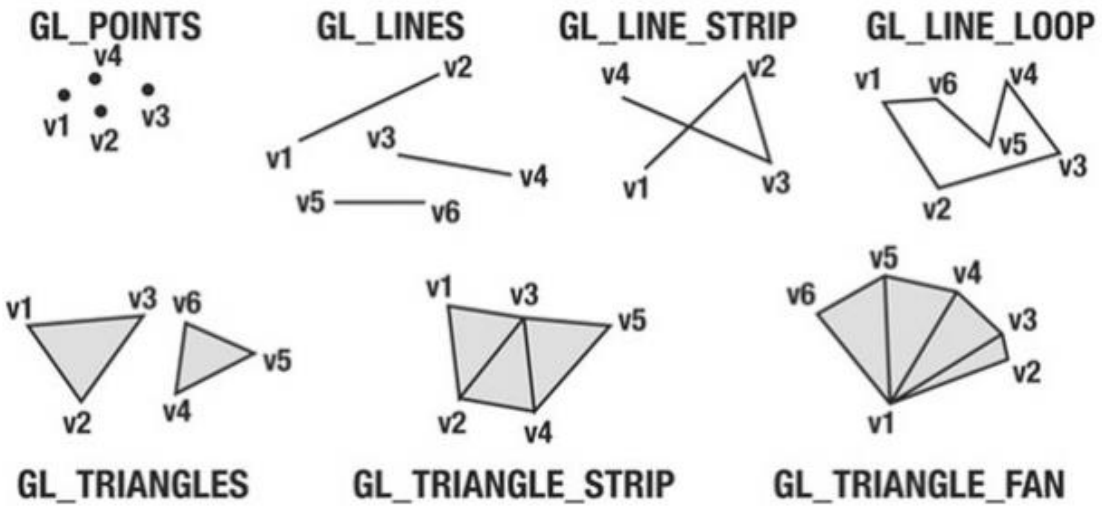
```
public void draw(GL10 gl) {

      // koska käytetään kärkinuolia, enabloidaan ne
      gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

      // laitetaan värit RGBA arvoina
      gl.glColor4f(1.0f, 0.0f, 0.0f, 1.0f);

      gl.glVertexPointer(
                coords_per_vertex,
                GL10.GL_FLOAT, 0, vertexBuffer);
      gl.glDrawArrays(
                GL10.GL_TRIANGLES, 0,
                triangleCoords.length / coords_per_vertex);

      // Disabloidaan nuolikärkien piirto, jotta ei tule konflikteja muiden
      // muotojen kanssa
      gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
}
```

Esimerkkikoodi 4. Kolmion piirto ruudulle.

Esimerkkikoodin 4 gl.glVertexpointer()-metodi kertoo OpenGL ES:lle verteksien positiot. Metodin ensimmäinen parametri kertoo, että jokaisella verteksillä on 3 koordinaattia. Toinen parametri kertoo datatyypin, joka tässä tapauksessa oli float. Kolmas parametri on ns. stride, joka kertoo tavuina, kuinka kaukana jokaisen verteksin sijainti on toisistaan. Tässä tapauksessa verteksit olivat samassa koordinaatistossa, eli hyvin lähellä toisiaan, eli sen voi laittaa arvoon 0. Viimeinen parametri on sisältää luodun bufferin. Lopuksi

piirretään kolmio gl.glDrawArrays()-metodilla. Siinä ensimmäisenä parametrina annetaan muodon tyyppi, mitä lähdetään piirtämään. Toisena parametrina annetaan sen kolmion positio, mitä lähdetään piirtämään. Kolmantena annetaan piirrettävien verteksien lukumäärä.

Kolmiot eivät suinkaan ole ainoita muotoja, joita OpenGL ES suostuu rasteroimaan. Kuvassa 4 nähdään kaikki alkumuodot, joita OpenGL ES tarjoaa kehittäjälle.

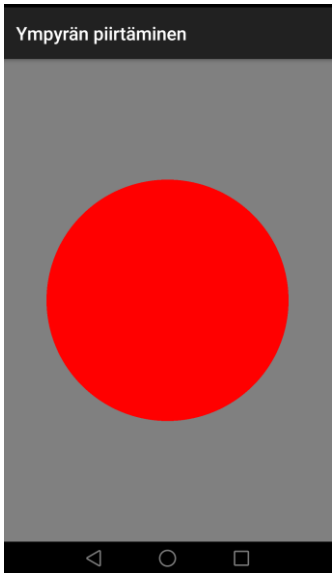



Kuva 4. OpenGL ES:n tarjoamat muodot. [14, s. 325]

Katsotaan jokaista hieman tarkemmin:

- Points – Jokainen verteksi (v) on oma yksilönsä.
- Lines – Viiva koostuu kahdesta eri verteksistä, jotka on yhdistetty viivalla.
- Line_strip – Voi tehdä x-määrästä verteksejä, joista muodostuu yksi pitkä viiva.
- Line_loop – Sama kuin line_strip, sillä erotuksella, että OpenGL ES automaattisesti piirtää viivan aloitus- ja lopetusverteksin välille.
- Triangle – Kolmion piirtäminen.
- Triangle_strip - Sen sijaan, että määritettäisiin kolme pistettä, määritämme vain kolmioiden määrät + 1 verteksi. OpenGL ES muodostaa tällöin ensimmäisen kolmion vertekseistä (v1,v2,v3). Seuraava kolmio muodostuu vertekseistä (v2,v3,v4) ja niin edelleen.
- Triangle_fan – Tässä on yksi pääverteksi (v1) joka on yhteinen kaikkien muiden verteksien kanssa. Ensimmäinen kolmio on (v1,v2,v3), seuraava on (v1,v3,v4) ja niin edelleen.

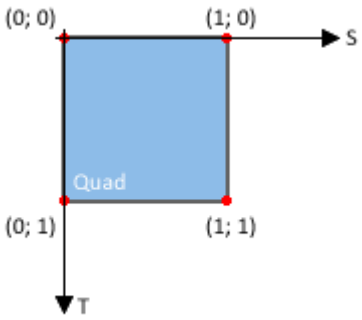
Esimerkiksi GL_TRIANGLE_FAN-muodolla pystyy piirtämään hienon ympyrän. Sen voi toteuttaa niin, että on yksi pääverteksi, joka sijaitsee ympyrän keskellä. Siitä halutun välimatkan päässä on viivan toinen pääty eli toinen verteksi. Näitä viivoja voi sitten piirtää 360 niin, että pääverteksi pysyy aina samana, mutta uuden viivan toinen pääty siirtyy aina asteen verran eteenpäin, kuitenkin niin, että viivan pituus pysyy samana. Näin lopputuloksena on kuvan 5 mukainen ympyrä.




Kuva 5. Ympyrän piirtäminen.

### 3.1.2 Tekstuurit

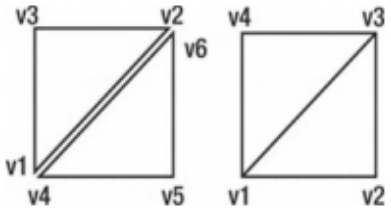
Jotta kuitenkin päästäisiin hieman hienompiin sovelluksiin, niin voidaan myös piirtää tekstuureja. OpenGL ES perustuu hyvin paljon kolmioiden piirtoon, joten tekstuurit voidaan myös piirtää kolmioiden sisälle. Tämä tapahtuu helposti antamalla jokaiselle verteksille uudet attribuutit, jotka kuvastavat kyseisen verteksin tekstuurikoordinaatteja. Tekstuurikoordinaatit eroavat hieman verteksien kuvan 3 koordinaattisysteemistä. Vertekseillähän on x- ,y- ja z- koordinaatit, OpenGL ES kutsuu tekstuurien koordinaatteja s ja t. S vastaa x-koordinaattia ja t vastaa y-koordinaattia. Kuten kuvasta 6 huomataan, t-koordinaatti menee ylösalaisin, eli suurenee alaspäin mentäessä. Tämä on otettava huomioon, kun laittaa vertekseille tekstuurikoordinaatteja.




Kuva 6. Tekstuurien koordinaattisysteemi. [15]

Tekstuurin eli kuvan vasen yläkulma tulee aina olemaan kohdassa 0,0, ja oikea alakulma kohdassa 1,1. Vaikka kuvan leveys olisi kaksi kertaa niin suuri kuin korkeus, niin kuvan kulmat ovat silti edellä mainitussa kohdassa. Näitä kutsutaan normalisoiduksi koordinaateiksi [14, s. 305].

Jos halutaan piirtää neliön muotoiseen alueeseen jokin tekstuuri, niin voi tehdä kaksi kolmiota vierekkäin, jotka muodostavat neliön kuvan 7 mukaisesti.




Kuva 7. Vasemmalla neliön piirtäminen kuudella verteksillä, ja oikealla neljällä verteksillä. [14, s.315]

Kuvassa 7 on piirretty vasemmalla kaksi kolmiota, jossa on kuusi verteksiä. Tästä huomataan, että verteksien v1 ja v2 muodostama viiva piirrettäisiin uudestaan vertekseillä v4 ja v6. Tämän sijasta on hyvä poistaa päällekkäisyydet ja määrittää verteksi v2 ja v6 vain yhtenä verteksinä v3, ja verteksi v1 ja v4 verteksinä v1. Tämä rasteroi silti kaksi kolmiota, mutta voimme kertoa OpenGL ES:lle, mitä verteksejä käytetään kolmioiden piirtämiseen (ensimmäiseen kolmioon verteksit v1,v2 ja v3, toiseen kolmioon verteksit v3,v4 ja v1). Tämä voidaan tehdä indeksöimällä verteksit. Kuvan 7 oikeanpuolen neliön verteksi v1 on indeksi 0, v2 indeksi 1 ja niin edelleen. Kuvan 7 neliöstä saataisiin esimerkkikoodin 5 mukainen indeksitaulukko.

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(6 * 2);
byteBuffer.order(ByteOrder.nativeOrder());
indices = byteBuffer.asShortBuffer();

indices.put(new short[] { 0, 1, 2,
                          2, 3, 0 });
```

Esimerkkikoodi 5. Indeksitaulukko, joka laitetaan bufferiin.

Kuten huomataan, esimerkkikoodi 5:ssa tehdään myös verteksien tapaan javan bufferi, johon tiedot kirjoitetaan. OpenGL ES vaatii indekseille kuitenkin short-tyyppisen bufferin floatin sijasta. Yksi short vie 2 tavua muistia, joten muistia täytyy varata 12 tavua (6 indeksiä, joista jokainen vie 2 tavua).

Jotta tekstuuri saataisiin piirrettyä kahden kolmion muodostamaan neliöön käyttäen määrittämiämme indeksejä, tarvitaan verteksit, mihin on lisäksi määritetty tekstuurikoordinaatit.

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(16 * 4);
byteBuffer.order(ByteOrder.nativeOrder());
vertices = byteBuffer.asFloatBuffer();
// luodaan 4 verteksiä, joilla x, y, s ja t koordinaatit
vertices.put(new float[] {  0.2f, 0.2f, 0.0f, 1.0f,
                            0.8f, 0.2f, 1.0f, 1.0f,
                            0.8f, 0.6f, 1.0f, 0.0f,
                            0.2f, 0.6f, 0.0f, 0.0f }
);
```

Esimerkkikoodi 6. Neljä verteksiä, jossa mukana tekstuurikoordinaatit.

Koska käytetään indeksejä, niin tarvitaan vain neljä verteksiä kahden kolmion piirtämiseen. Esimerkkikoodissa 6 luodaan vertekseille niille tarvittava float bufferi, johon verteksit laitetaan. Tilaa varataan 16 * 4 tavun verran, koska koordinaatteja on yhteensä 16. Sillä ei ole väliä, onko se tekstuuri vai verteksi koordinaatti, se vie silti 4 tavua muistia. Taulukon ensimmäinen verteksi saa x- ja y- koordinaattien arvoksi 0.2 (tehdessä 2D-kuvaa, z koordinaatin voi jättää pois, jolloin OpenGL ES automaattisesti pistää sen arvoon 0) ja tekstuurikoordinaatiksi ensimmäinen verteksi saa s:n arvoksi 0,0 ja t: n arvoksi 1.0. Koska hyvin muistamme, niin t-koordinaatti meni tekstuurikoordinaatistossa väärinpäin, niin arvot (0,0 , 1,0) viittaavat tekstuurikoordinaatiston vasempaan alakulmaan. Koska aikaisemmin laitoimme indeksit alkamaan nollasta (verteksi v1), niin silloin täytyy ensimmäisen verteksin sijainti olla kuvan 7 mukaisesti vasemmassa alakulmassa. Toinen verteksi saa x:n ja y:n arvot 0.8 ja 0.2, eli laitamme neliölle leveyttä siirtämällä toisen verteksin x:n arvoa oikealle. Tekstuurikoordinaatit (1,0 , 1,0) viittaavat

neliön oikeaan alakulmaan, mikä tarkoittaa indeksiä 1. Samalla tyylillä tehdään myös verteksit v3 ja v4, jolloin neliö muodostuu.

Itse piirtäminen onnistuu esimerkkikoodin 7 mukaisesti.

```
//Laitetaan verteksit ja tekstuurien koordinaatit päälle
gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

//Laitetaan 2D tekstuurien piirto päälle
gl.glEnable(GL10.GL_TEXTURE_2D);

vertices.position(0);
gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
vertices.position(2);
gl.glTexCoordPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);

gl.glDrawElements(GL10.GL_TRIANGLES, 6, GL10.GL_UNSIGNED_SHORT, indices);
```

Esimerkkikoodi 7. Kuvan piirtäminen ruudulle.

Jotta tekstuuri saataisiin piirrettyä, pitää ensin laittaa tarvittavat tilat päälle esimerkkikoodin 7 mukaisesti. Tämän jälkeen laitetaan verteksin koordinaattien luku alkamaan arvosta 0, eli ensimmäisestä koordinaatista. GlVertexPointer()-metodilla kerrotaan OpenGL ES:lle että kaksi ensimmäistä koordinaattia ovat verteksikoordinaatteja, ne ovat tyyppiä float, kerrotaan koko verteksin tavujen koko (4 koordinaattia, eli 16 tavua) ja lopuksi laitetaan luotu verteksien bufferi. Tämän jälkeen laitetaan position()-metodilla verteksikoordinaattien luku alkamaan kolmannesta koordinaatista, mikä oli ensimmäinen tekstuurikoordinaatti. Samaan tyyliin glTexCoordPointer()-metodilla luetaan kyseiset kaksi verteksin tekstuurikoordinaattia. Itse piirtäminen onnistuu glDrawElements()-metodilla, joka on hyvin samankaltainen glDrawArrays()-metodin kanssa. Ensimmäinen parametri kertoo, että piirretään kolmioita. Toinen on indeksien määrä, mikä tässä tapauksessa on 6. Kolmas parametri kertoo indeksien tyypin, joka on short, ja viimeinen parametri on viittaus indekseihin.

Oleellista tietenkin on itse tekstuuritiedoston lataus, jotta OpenGL ES voi käyttää sitä neliössämme. Katsotaan hieman esimerkkikoodia 8, joka lataa tekstuurin sovelluksen käyttöön

```
try {
      Bitmap bitmap = BitmapFactory.decodeStream(
      game.getFileIO().readAsset("yourImage.png"));

      int textureIds[] = new int[1];
      gl.glGenTextures(1, textureIds, 0);
```

```
        int textureId = textureIds[0];

        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);

        //Säädetään OpenGL ES: än käyttämät filterit.
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,
        GL10.GL_NEAREST);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER,
        GL10.GL_NEAREST);

        gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
        bitmap.recycle();

} catch(IOException e) {
        throw new RuntimeException("Tiedoston lataus epäonnistui ");
}
```

Esimerkkikoodi 8. Tekstuuritiedoston lataus ja käyttö.

BitmapFactory.decodeStream()-metodi ottaa annetun kuvatiedoston talteen Bitmap-muuttujaan. Koska OpenGL ES on C-kielen API, ei se suoraan voi käyttää tekstuuria, vaan se tarvitsee ID:n, jossa tekstuuri on. Tätä varten luodaan ensin 1-paikkainen int taulukko (koska halutaan ladata vain 1 tekstuuri), glGetTextures()-metodilla kerrotaan OpenGL ES:lle, että halutaan tehdä uusi tekstuuri. Metodin ensimmäinen parametri kertoo tehtävien tekstuurien määrän eli 1. Toinen parametri on viittaus taulukkoon, johon tekstuuri ladataan, ja kolmas parametri on vain tieto siitä, mistä kohtaa taulukkoa OpenGL ES alkaa tekstuureja lisäämään.

Jotta kuva saataisiin OpenGL ES:än liitettyä, pitää se sitoa siihen. Tämä tapahtuu glBindTexture()-metodilla (Koska halutaan 2D-tekstuuri, käytetään GL_TEXTURE_2D). Android-kehyksen oman luokan GLUutils texImage2D()-metodilla saadaan itse kuva liitettyä mukaan. Siihen parametreina sanotaan taas, että käytetään 2D-tekstuuria, ja myös aikaisemmin lataamamme kuva annetaan parametrina. Tämän jälkeen OpenGL ES käyttää kaikissa tekstuuriin liittyvissä kutsuissa kyseistä kuvaa. Kun kuva on määritelty, niin sen jälkeen sen sitominen on hyvä ottaa pois. Tämä tapahtuu samalla glBindTexture()-metodilla. Siinä tällä kertaa toinen parametri on vain laitettu arvoon 0. Tämä automaattisesti kertoo OpenGL ES:lle ottaa sitominen pois. Lopuksi viellä bitmapin recycle()-metodilla poistetaan kuva muistista.


Kuva 8. Tekstuuri piirrettynä OpenGL ES 1.x:ssä.

### 3.1.3 Projektiot ja matriisit

OpenGL ES määrittelee kaiken 3D-avaruudessa. Jotta siitä päästäisiin 2D-avaruuteen, tarvitaan projektioita. Kuten aikaisemmin mainittu on, OpenGL ES koskee pääasiassa kolmioita, joilla on kolme pistettä 3D-avaruudessa. Jotta kolmio voidaan muodostaa kehyspuskurissa, OpenGL ES:n on tunnettava näiden 3D-pisteiden koordinaatit pikselipohjaisessa koordinaatistossa. Kun se tietää nämä kolme kulmapisteen koordinaatistoa, se voi yksinkertaisesti piirtää pikselit kehyspuskuriin, jotka ovat kolmion sisällä.

Jotta piirretystä kolmiosta saataisiin kaksiulotteinen, voidaan käyttää ortografista projektiota. Se muodostaa kolmiulotteisesta kolmiosta kaksiulotteisen kuvan ilman syvyysvaikutelmaa. Tällöin kuvan 3 koordinaatistossa syvyysarvo jää nollaksi. [14, s. 272]

OpenGL ES ilmaisee projektioita matriisien muodossa. Kehittäjän kannalta ei ole tarpeen tietää matriisien sisäosia, vaan on vain tiedettävä, mitä eri matriisit tekevät pisteissämme, jotka kehittäjä on määrittänyt koordinaatistoon. OpenGL ES käyttää kolmea erillaista matriisia, joita se hyödyntää määrittelemissämme pisteissä:

- Model-view matrix: Tätä voi käyttää kolmion pisteiden siirtoon, rotaatioon (kiertää pisteitä origon kautta kulkevan akselin ympäri) ja skaalaukseen (suurentaa pisteitä kertomalla niiden koordinaatit jollakin luvulla). Tätä matriisia käytetään myös määrittämään kameran sijainti ja suunta.
- Projection matrix: Tämä matriisi projisoi kolmiulotteiset koordinaatit kaksiulotteiselle näyttöruudulle.
- Texture matrix: Tämä matriisi antaa kehittäjälle mahdollisuuden manipuloida teksturointikoordinaatteja.

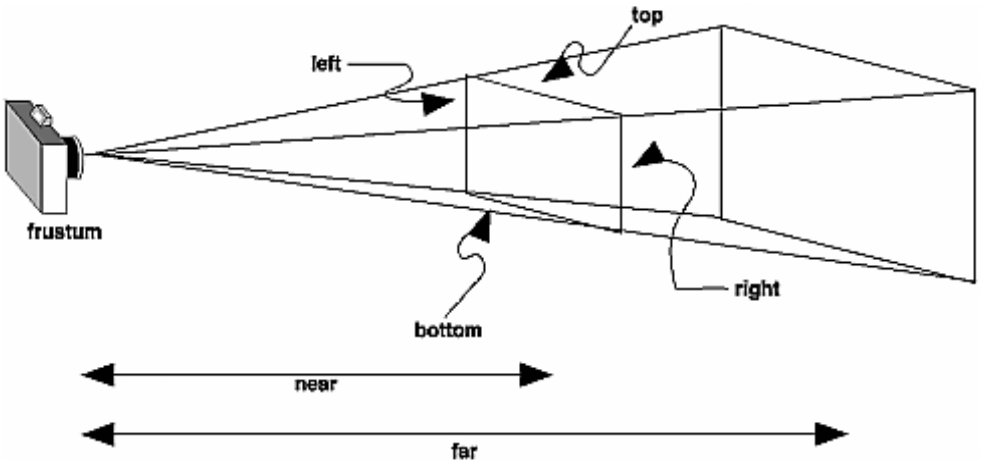
Projektio suoritetaan OpenGL ES 1.x API:ssa onSurfaceChanged()-metodissa. Aina kuin kohdelaitteen näyttöä muutetaan vaikka laittamalla se vaakatasoon, suoritetaan kyseinen metodi, jossa käytetään projektiota esimerkkikoodin 9 tavalla.

```
public void onSurfaceChanged(GL10 gl, int width, int height){

      gl.glViewport(0, 0, width, height);
      // Tekee näytön suhteet oikeaksi
      float ratio = (float) width / height;
      // Koska tehdään 2D-muunnos, käytetään projektio matriisia
      gl.glMatrixMode(GL10.GL_PROJECTION);
      // resetoi matriisin sen oletus tilaan
      gl.glLoadIdentity();
      //Käytetään projektio matriisia
      gl.glFrustumf(-ratio, ratio, -1, 1, 3, 7);

}
```

Esimerkkikoodi 9. Projektiomatriisia käytetty.

gl.gllViewport()-metodin kaksi ensimmäistä parametria määrittää näkökentän ylä-vasen-kulman koordinaatit. Width- ja height-parametrit taas kertovat näytön leveyden ja korkeuden pikseleinä. Gl.glMatrixMode()-metodissa määritetään se projektion tyyppi, mitä halutaan käyttää. Jos kehittäjä haluaa tehdä 2D-muunnoksen, niin on hyvä käyttää projektiomatriisia (GL10.GL_PROJECTION). Jos haluaa tehdä aikaisemmin mainittuja kahta muuta temppua, niin voi käyttää joko GL10.GL_MODELVIEW tai GL10.GL_TEXTURE-matriiseja. Kaikki manipulaatiot tämän jälkeen kohdistuvat tähän säädettyyn matriisityyppiin. Jos haluaa tehdä manipulaatioita esimerkiksi tekstuureihin, niin sitä ennen on gl.glMatrixMode()-metodissa vaihdettava matriisityyppi tekstuuriksi. Tätä matriisin tyyppiä voidaan kutsua yhdeksi OpenGL ES:n tiloista. Kuten aikaisemmin on mainittu, kaikki tilat eli konteksti häviää, kun sovellus on laitettu tauolle ja siitä takaisin päälle. Kehittäjän on siis oltava huolellinen, että tämä kontekstin häviö otetaan ohjelmakoodissa huomioon.

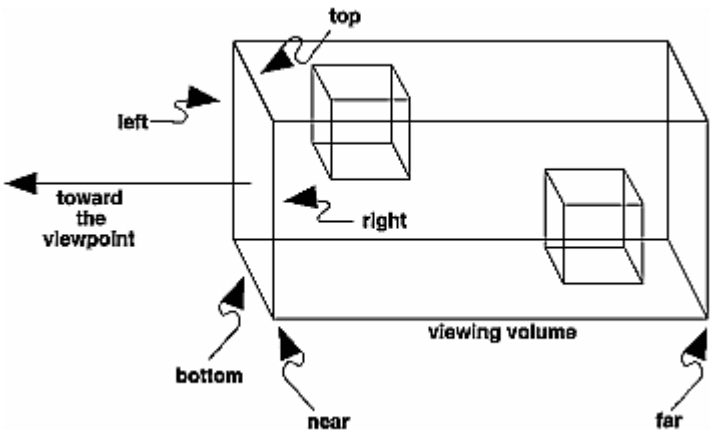
Mielenkiintoinen on esimerkkikoodissa 9 käytetty glFrustumf(double left, double right, double bottom, double top, double near, double far)-metodi. Siinä määritetään se näkökenttä, mikä sovellukseen tulee näkymään. Voisi ajatella, että se näkymä, mikä silmällä katseltaessa tai kameralla kuvattaessa saadaan, tulee näkökenttään kuvan 9 mukaisesti.




Kuva 9. glFrustum()-metodin perspektiivinäkymä. [16]

Tätä näkymää kutsutaan perspektiivinäkymäksi. Perspektiivi tekee kauempana olevista objekteista pienempiä kuin ne, mitkä ovat lähempänä silmää (kameraa). Tätä projisointimenetelmää käytetään yleisesti animaatioon, visuaaliseen simulointiin ja muihin sovelluksiin, jotka pyrkivät jonkinasteiseen realismiin, koska se toimii samalla tavalla kuin silmä (tai kamera). [16.]

Toinen mahdollinen projisointitapa on aikaisemmin mainittu ortografinen projektio. Se tekee näkökentästä laatikon mallisen kuvan 10 tapaan. OpenGL ES:sä sen voi toteuttaa metodilla glOrthof(int left, int right, int bottom, int top, int near, int far). Toisin kuin perspektiivinäkymässä, näkökentän suuruus ei vaihdu toisesta päästä toiseen, eli objektit näyttävät näkökentässä yhtä suurena - oli kameran etäisyys sitten mikä tahansa. Tällöin 2D-ulottovuutta tehdessä voi metodin near- ja far- arvot pistää lähelle nollaa (on hyvä jättää edes vähän z-koordinaatin etäisyyttä, jotta se saa vähän bufferointi tilaa) [14, s. 291]. Tätä projisointimenetelmää käytetään sovelluksissa, jotka tekevät esimerkiksi arkkitehdin pohjapiirrustuksia, jossa on tärkeää ylläpitää objektien ja kulmien välisiä todellisia kokoja niiden projisoidessa.




Kuva 10. glOrthof()-metodin ortografinen projektio. [16]

Käytettäessä glOrthof()-metodia voidaan sillä määrittää myös näkökentän koon niin kuin sen haluaa. Jos esimerkiksi metodille laitetaan parametrit glOrthof(0, 480, 0, 320, 1, -1), niin laatikon etupuolen (kameraan päin olevan) vasen alakulma tulee samaksi kuin näkökentässä oleva. Tässä tapauksessa arvoon 0,0. Oikea yläkulma tulee myös samaksi eli 480, 320. Eli tällöin näkökentän (se mikä sovelluksen ruudulle tulee) koordinaatit ovat x-akselilla väliltä 0 ja 480, ja y-akselilla väliltä 0 ja 320. Tätä voi sitten käyttää, kun esimerkiksi piirtää sovellukseen verteksejä, niin niille annettavat koordinaatit täytyy antaa kyseisiltä koordinaattiväleiltä.

Kun projektiomatriisi on tehty, ja näytön objektin uuden koordinaatit on laskettu, pitää vielä soveltaa kameranäkymää. Esimerkkikoodi 10 näyttää, miten onDrawFrame()-metodia käytetään niin, että se simuloi kameran sijaintia.

```
public void onDrawFrame(GL10 gl) {
      // Laitetaan modelview matriisi käyttöön
      gl.glMatrixMode(GL10.GL_MODELVIEW);
      gl.glLoadIdentity();   // resetoi matriisin sen oletus tilaan
      // Laitetaan kameran sijainti kohdilleen
      GLU.gluLookAt(gl, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

}
```

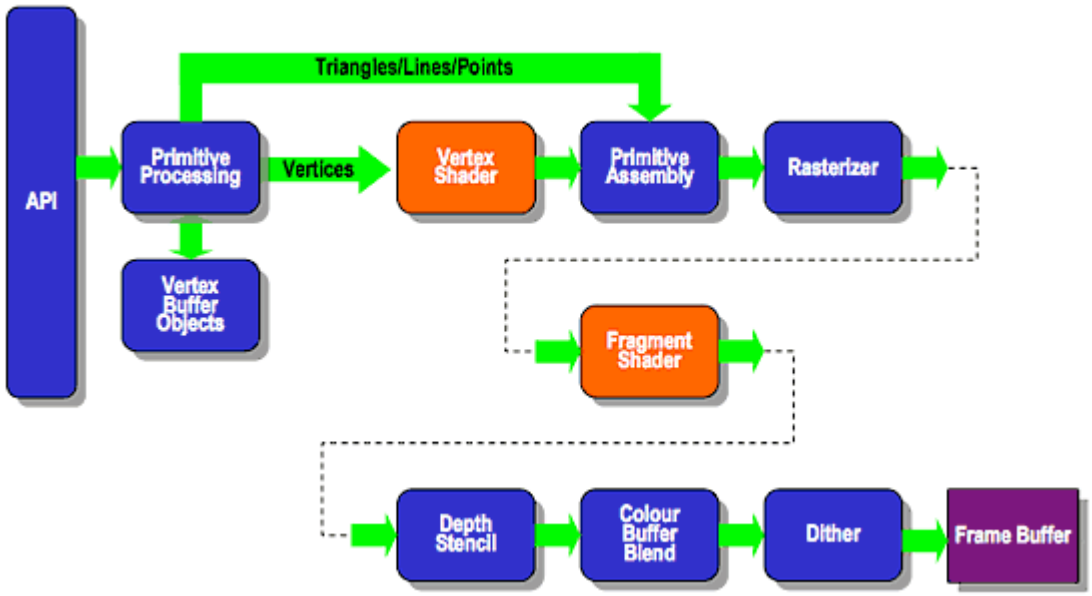
Esimerkkikoodi 10. Kameranäkymän luonti.

Jotta kuva saadaan näkymään, on vielä määritettävä kameran sijainti. Tämä tapahtuu gluLookAt()-metodissa. Siinä määritetään kameran (silmän) sijainti, näkökentän keskikohta sekä ylöspäin menevä vektori. Tätä tehdessä on GL_MODELVIEW-matriisin

oltava käytössä. Näin saadaan kamera sijoitettua siihen kohtaan, josta näkökenttää halutaan katsoa, eli ruudulle tulee kuva siitä kuvakulmasta kuin halutaan.

## 3.2 OpenGL ES 2.0

OpenGL ES 2.0 julkaistiin julkisesti maaliskuussa 2007. Se eroaa edeltäjästään hyvinkin paljon, sillä siinä OpenGL 1.x:n omaava kiinteän funktion putki on korvattu kokonaan kuvan 11 näköisellä ohjelmoitavalla putkella. Tämä on toteutettu varjostimien avulla (shaders).




Kuva 11. OpenGL ES 2.0:n ohjelmoitava putki [12].

Kuvasta 11 huomataan, että osa OpenGL ES 1.x:n putken (kuva 2) vaiheista on poistettu, ja korvattu varjostimilla. Transform and Lightning -vaihe on korvattu kokonaan verteksivarjostimilla. Texture Environment-, Colour Sum- ja For-vaiheet on korvattu fragmenttivarjostimella. OpenGL ES 2.0 ohjelmoitava putki siis käyttää näitä kahta varjostimen tyyppiä, jotta laitteeseen saadaan piirto näkymään ruudulle.

OpenGL ES 2.0 -varjostimet ovat kuvan piirtoprosessin vaiheet, joita kehittäjä pääsee vapaasti muokkaamaan. Ne antavat kehittäjille mahdollisuuden luoda uusia piirtotehosteita, ja täten tehdä ohjelmista näyttävämmän näköisiä. Jokainen OpenGL ES 2.0-sovellus vaatii aikaisemmin mainitut kaksi varjostinta: verteksi ja

fragmenttivarjostimet. OpenGL ES käyttää varjostimissa ihan omaa sille tarkoitettua varjostimen kieltä nimeltä OpenGL ES Shading Language (GLSL ES). Varjostimia tehdessä siis kirjoitetaan GLSL ES -kielellä oma pieni suoritettava koodinpätkä, joka käsittelee pikselien esittämisen näyttöön tietyn datan avulla sovelluksestasi. OpenGL ES 2.0 käyttää GLSL ES -versiota 1.00. Kehittäjille näiden tekeminen tuo hiukan enemmän haastetta, jolloin OpenGL ES 2.0 -sovelluksen tekeminen on hieman monimutkaisempaa kuin OpenGL ES 1.x -sovelluksen tekeminen.


Kuva 12. Varjostimilla ja ilman varjostimia tuotettu kuva [17].

### 3.2.1 Muotojen piirtäminen verteksivarjostimien avulla

Verteksivarjostimet ovat vakiintuneimpia ja yleisimpiä 3D-varjostinmuotoja. Se suoritetaan yksi kerrallaan jokaiselle varjostinprosessiin annetulle verteksipisteelle. Verteksivarjostimilla voi manipuloida verteksien ominaisuuksia, kuten sijaintia, väriä, valaistusta ja tekstuurikoordinaatteja. Kun muunnokset on tehty vertekseille, annetaan lopputulos seuraavalle putken vaiheelle.

Verteksivarjostin muodostuu GLSL ES -kielen koodinpätkästä, joka ajetaan ja suoritetaan varjostinprosessissa. Tutkitaan hieman esimerkkikoodia 11.

```
public static final String vertexShaderCode =

     "uniform mat4 uMVPMatrix;" +
     "attribute vec4 vPosition;" +
     "void main() {" +
     " gl_Position = uMVPMatrix * vPosition;" +
     "} ";
```

Esimerkkikoodi 11. Verteksivarjostimen luonti.

Esimerkkikoodissa 11 luodaan pieni vertekstivarjostimen koodinpätkä, joka muodostaa verteksien koordinaatit normalisoituun muotoon. Siinä on kaksi parametria, joita se vähintään tarvitsee. uMVPMatrix (Model View Projection Matrix) on tyyppiä mat4, eli 4x4-ulotteinen matriisi. Se pitää sisällään transformaatiomatriisin, jota voidaan käyttää muuttamaan pikselikoordinaatin kohdelaitteelle sopivaan koordinaattiin (normalisoituun muotoon). Se on myös uniform, eli ne ovat vakioita ja yhtenäisiä kaikissa sovelluksen varjostimissa.

Toinen parametri vPosition on attribuuttityyppiä vec4. Attribuutit ovat jokaiselle verteksille omia, ja niiden näkyvyys on vain siihen verteksivarjostimeen, johon se on luotu. Tässä esimerkissä on luotu verteksi (vec4), jossa on neljä eri koordinaattia (x,y,z,w).

Tämän jälkeen tulee main()-metodi, joka pitää myös olla, jotta verteksivarjostimen koodi suoritetaan. Siinä luodaan gl_Position-niminen muuttuja, joka kertoo verteksin normalisoidun koordinaatin. Se tapahtuu kertomalla 4x4-ulotteinen matriisi (uMVPMatrix) verteksin pikselikoordinaatilla (vPosition). Jotta ymmärtäisimme paremmin, mitä tämän kertolaskun takana oikein tapahtuu, katsotaan kuvaa 13.

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

Kuva 13. Matriisi, jolla tehdään vertekseille transformaatioita [18].

Voidaan ajatella, että kun halutaan saada uusi positio verteksille (x,y,z,w), niin esimerkiksi uusi x-positio saadaan kertomalla matriisin ensimmäinen rivi (a,b,c,d) jokaisen verteksin koordinaatilla. Tällöin kun ne lasketaan yhteen, saadaan verteksille uusi x. Uusi y-positio saadaan kertomalla matriisin toinen rivi (e,f,g,h) verteksin jokaisella koordinaatilla, ja niin edelleen. Huomioitavaa tässä on, että verteksillä on oikeastaan 4 koordinaattia. Leveys (x), korkeus (y), syvyys (z) ja viimeisenä on w muuttuja. Jos tämä on 1, niin verteksi ajatellaan positiona avaruudessa. Jos se on 0, niin verteksi ajatellaan suuntana. [18.] Yksinkertaisena esimerkkinä voidaan katsoa hieman käännösmatriisin tekoa.

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Kuva 14. Käännösmatriisi [18]

Kuvan 14 käännösmatriisissa x, y ja z ovat arvot, jotka halutaan lisätä verteksin positioon.

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 10 \\ 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*10+0*10+0*10+10*1 \\ 0*10+1*10+0*10+0*1 \\ 0*10+0*10+1*10+0*1 \\ 0*10+0*10+0*10+1*1 \end{bmatrix} = \begin{bmatrix} 10+0+0+10 \\ 0+10+0+0 \\ 0+0+10+0 \\ 0+0+0+1 \end{bmatrix} = \begin{bmatrix} 20 \\ 10 \\ 10 \\ 1 \end{bmatrix}$$

Kuva 15. Matriisin ja vektorin kertolasku, jossa muutetaan vektorin sijaintia [18].

Jos halutaan siirtää vektoria (10,10,10,1), oikealle vaakatasossa 10 pikselin verran, saadaan kuvan 15 näköinen laskutoimitus. Tässä kerrotaan jokainen rivi vektorin koordinaateilla, jolloin lopputulokseksi saadaan verteksille uusi sijainti (20,10,10,1). Koska tehtiin position muunnos, niin verteksin w arvo on tässä 1. Huomioitavaa tässä kaikessa on se, että kun verteksivarjostimen gl_Position luodaan, niin juuri tämän kertolaskun takia on tärkeää, että kertolasku suoritetaan juuri uMVPMatrix * vPosition-järjestyksessä. Jos kertolaskun laittaa toisinpäin, niin lopputulos ei ole oikea.

Verteksivarjostimen käyttö vaatii että sovellukseen tehdään piirrettävälle muodolle myös projektiot ja kameranäkymät. OpenGL ES 1.x:n tapaan projektio määritellään onSurfaceChanged()-metodissa esimerkkikoodin 8 tapaan. Koodin alussa luodaan myös

16-paikkainen mMVPMatrix float-taulukko, johon tullaan määrittämään projektio- ja näkymämatriisit. Taulukot ovat 16-paikkaisia, koska 4x4-matriisissa on 16 alkiota.

```
private final float[] mMVPMatrix = new float[16];
private final float[] mProjectionMatrix = new float[16];
private final float[] mViewMatrix = new float[16];

public void onSurfaceChanged(GL10 unused, int width, int height) {

     GLES20.glViewport(0, 0, width, height);
     float ratio = (float) width / height;

Matrix.frustumM(mProjectionMatrix, 0, -ratio, ratio, -1, 1, 3, 7);
}
```

Esimerkkikoodi 12. Projektion määrittäminen OpenGL ES 2.0:ssa.

Esimerkkikoodi kahdessatoista käytetään OpenGL-kirjaston Matrix-luokkaa. Kyseistä luokkaa ei tarvitse alustaa, vaan sen staattisia metodeja voi käyttää. Matrix.frustum()-metodissa käytetään luomaamme mProjectionMatrix-taulukkoa, johon matriisi tehdään.

Projektion lisäksi kameranäkymä pitää määrittää. Se tehdään onDrawFrame()-metodissa esimerkkikoodin 13 tapaan.

```
public void onDrawFrame(GL10 unused) {

     Matrix.setLookAtM(mViewMatrix, 0, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

     Matrix.multiplyMM(mMVPMatrix, 0, mProjectionMatrix, 0, mViewMatrix, 0);

     triangle.draw(mMVPMatrix);
}
```

Esimerkkikoodi 13. Kameranäkymän ja lopullisen matriisin luonti.

Näkymämatriisi luodaan esimerkkikoodi yhdeksässä setLookAtM()-metodissa, jossa taas täytetään aikaisemmin tekemäämme mViewMatrix-matriisitaulukkoa. multiplyMM()-metodissa yhdistetään projektio ja näkymämatriisit, josta saadaan lopputuloksena transformaatiomatriisi, joka annetaan parametrina sille muodolle, joka halutaan piirtää.

Työn OpenGL ES 1.x osiossa tehtiin kolmioluokka, jossa tehtiin kolmion kolme eri pisteverteksiä ja laitettiin ne bufferiin. OpenGL ES 2.0:ssa voi tehdä samat toimenpiteet, mutta siihen pitää lisätä esimerkkikoodin 14 osoittamat toimenpiteet.

```
int vertexShader = MyGLRenderer.loadShader(
          GLES20.GL_VERTEX_SHADER, vertexShaderCode);
```

```
//Tekee tyhjän OpenGL ohjelman
mProgram = GLES20.glCreateProgram();
//Lisää siihen tekemämme verteksivarjostimen
GLES20.glAttachShader(mProgram, vertexShader);
//Linkitetään ohjelma objecti
GLES20.glLinkProgram(mProgram);
```

Esimerkkikoodi 14. OpenGL ES -ohjelman luonti.

Esimerkkikoodin 14 alussa ladataan esimerkkikoodin 11 verteksivarjostin, jotta sitä voidaan käyttää. Sitten tehdään tyhjä OpenGL-ohjelma objekti, johon lisätään tekemämme verteksivarjostin GLE20.glAttachShader()-metodilla. Lopuksi ohjelma pitää linkittää GLES20-ympäristöön, jotta sitä voidaan käyttää.

Kolmion piirto ruudulle tapahtuu OpenGL ES 1.x: n tapaan draw()-metodissa. OpenGL ES 2.0:ssa metodin toteutus on jonkin verran monimutkaisempaa toteuttaa, kuten esimerkkikoodi 15 näyttää.

```
public void draw(float[] mvpMatrix) {
      // Käytetään tekemäämme ohjelma objektia
      GLES20.glUseProgram(mProgram);

      //Otetaan ohjelman objektin verteksivarjostin koodista vPosition muuttuja
      mPositionHandle = GLES20.glGetAttribLocation(mProgram, "vPosition");

      // Enabloidaan ohjelman verteksien käyttö
      GLES20.glEnableVertexAttribArray(mPositionHandle);

      // Valmistellaan kolmion koordinaatit
      GLES20.glVertexAttribPointer(
             mPositionHandle, 3,
             GLES20.GL_FLOAT, false,
             12, vertexBuffer);

      // Otetaan ohjelma objektista käyttöön uMVPMatrix matriisi
      mMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");

      // Lisätään aikaisemmin tekemämme transformaatio matriisi
      GLES20.glUniformMatrix4fv(mMVPMatrixHandle, 1, false, mvpMatrix, 0);

      // Piirretään kolmio
      GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

      // Otetaan verteksien käyttö pois päältä
      GLES20.glDisableVertexAttribArray(mPositionHandle);
}
```

Esimerkkikoodi 15. Kolmion piirtäminen OpenGL ES 2.0:ssa.

Esimerkkikoodissa 15 otetaan käyttöön verteksivarjostin koodin vPosition ja uMVPMatrix, joille tehdään tarvittavat toimenpiteet, jotta kolmio saadaan piirrettyä. Esimerkiksi GLES20.glUniformMatrix4fv()-metodissa käytetään esimerkkikoodissa 13 luotua transformaatiomatriisia, joka lisätään verteksivarjostimeen. Tämän jälkeen, kun

kolmio piirretään, se käyttää kyseistä transformaatiomatriisia, ja piirtää sen niin, kuin se on määritelty.

### 3.2.2 Fragmenttivarjostimet

Sitten kuin verteksivarjostinprosessi ja siitä seurannut rasterointiprosessi on suoritettu, tulee vuoroon fragmenttivarjostimet. Ne ovat fragmenttiprosessissa jokaista pikseliä (fragmenttia) kohti ajettavia varjostimia. Niiden päätehtävä on määrittää pikselin väriarvo, mutta ne voivat määrittää myös erilaisia grafiikkaefektejä kuten tekstuurit, valaistuksen laskenta, varjostus, läpinäkyvyys, heijastus jne. Fragmenttivarjostin ajetaan jokaiselle pikselille yksi kerrallaan, jolloin ne eivät voi käsitellä viereisiä fragmentteja samaan aikaan. Ne eivät voi myöskään muuttaa pikseleiden sijaintia.

Verteksivarjostimien tapaan myös fragmenttivarjostimet koostuvat GLSL ES -kielen koodinpätkästä. Esimerkkikoodista 16 nähdään, minkälainen on hyvinkin yksinkertaisen fragmenttivarjostimen koodi.

```
public static final String fragmentShaderCode =
      "precision mediump float;" +
      "uniform vec4 vColor;" +
      "void main() {" +
      "  gl_FragColor = vColor;" +
      "}";
```

Esimerkkikoodi 16. Fragmenttivarjostimen koodi.

Verteksivarjostimen tapaan siinä täytyy olla main()-metodi, joka ajaa varjostimen koodin. gl_FragColor on sisäänrakennettu muuttuja, joka määrittää lopullisen värin fragmenttivarjostimelle. Esimerkkikoodin 16 tapauksessa se antaa kaikille pikseleille punaisen värin. Precision on tässä hyvinkin tärkeä arvo. Se määrittää tarkkuusasteen, mitä näytönohjain käyttää laskiessaan float-arvoja. Tähän on olemassa kolme eri vaihtoehtoa; lowp, mediump ja highp. Jos käyttää lowp:tä niin tarkkuus on heikohko, mutta rasterointi on nopeaa. Jos taasen on highp, niin tarkkuus on erinomainen, mutta rasterointi on hidasta. Jotkut laitteet eivät ollenkaan tue highp-tarkkuutta, eli niissä laitteissa ei sovellus pyöri ollenkaan, tai jos laite highp:tä tukeekin, niin siinä voi silti tulla suorituskykykapasiteetti vastaan. Tästä syystä olisi hyvä käyttää aina kun vain mahdollista mediump- tai lowp-tarkkuutta, jolloin taataan sovelluksen sulava toimivuus kaikissa laitteissa.

Kun fragmenttivarjostimen koodi on määritetty, niin pitää se verteksivarjostimen tapaan esimerkkikoodissa 14 luotuun OpenGL-ohjelmaobjektiin lisätä. Tämän jälkeen muotoa piirrettäessä otetaan ohjelma objektista vColor muuttujakäyttöön, ja GLES20.glUniform4fv()-metodilla lisätään siihen väri, jota se tulee käyttämään.

### 3.2.3 Tekstuurit

OpenGL ES 1.x:n tapaan tekstuureja tehdessä käytetään samaa s/t-koordinaattisysteemiä. Samalla tavalla OpenGL ES 2.0:ssa tehdään tekstuurikoordinaateille oma javabufferi, oma tekstuuri ID, johon ladattu kuva sidotaan. Kuvalle tehdään filteröinnit, ladataan GLUtils luokan textImage2D()-metodiin sidottu kuva ja lopuksi recycle()-metodilla poistetaan kuva muistista. Eroa OpenGL ES 1.x:än tulee, kun halutaan kuva piirtää ruudulle. Siinä käytetään jälleen varjostimia. Jos kuvan haluaa taas neliön muotoisen muodon sisälle piirtää, niin pitää tehdä neliölle oma varjostinohjelma, sekä tekstuureille pitää tehdä oma varjostinohjelma esimerkkikoodin 17 mukaan.

```
public static final String vertexShaderCode_Image =
      "uniform mat4 uMVPMatrix;" +
      "attribute vec4 vPosition;" +
      "attribute vec2 a_texCoord;" +
      "varying vec2 v_texCoord;" +
      "void main() {" +
      "  gl_Position = uMVPMatrix * vPosition;" +
      "  v_texCoord = a_texCoord;" +
      "}";

public static final String fragmentShaderCode_Image =
     "precision mediump float;" +
      "varying vec2 v_texCoord;" +
      "uniform sampler2D s_texture;" +
      "void main() {" +
      "  gl_FragColor = texture2D( s_texture, v_texCoord );" +
      "}";
```

Esimerkkikoodi 17. Tekstuurivarjostimen koodi.

Huomataan uusi attribuutti vec2 (verteksi kahdella koordinaatilla, niinkuin tekstuuri koordinaatissa kuuluu olla). Siihen laitetaan verteksivarjostimessa syöttönä tekstuurien koordinaatit a_textCoord-muuttujaan. Lisäksi on tehty varying-tyyppinen v_texCoord-niminen muuttuja, joka välittää tekstuurikoordinaattitiedot fragmenttivarjostimelle lineaarisen interpolaation avulla kolmion pinnan päällä [19]. Fragmenttivarjostimessa on uusi uniform-tyypin muuttuja sampler2D, joka kuvastaa

tekstuurin dataa. Gl_FragColor-muuttujassa laitetaan lopulliseen tuotokseen tekstuuri texture2D()-metodissa.

Kun varjostimista on tehty kaksi OpenGL-ohjelmaobjektia, niin GLES20.glVertexAttribPointer()-metodissa valmistellaan neliön verteksien koordinaatit verteksien bufferista, mutta nyt täytyy myös tekstuurikoordinaattien bufferista samalla metodilla valmistella niiden koordinaatit. Tämän jälkeen OpenGL ES 1.x:n tapaan GLES20.glDrawElements()-metodilla piirretään tekstuuri näytölle.

## 3.3 OpenGL ES 3.0/3.1/3.2

Vuonde 2012 elokuussa julkaistiin OpenGL ES versio 3.0. Se on taaksepäin yhteensopiva OpenGL ES 2.0 -version kanssa, joten sovellukset, jotka on kirjoitettu käyttäen OpenGL ES versiota 2.0, tulevat toimimaan myös laitteissa, jotka käyttävät versiota 3.0.

OpenGL ES 3.0 toi paljon uusia ominaisuuksia, joita kehittäjä pystyy hyödyntämään. Niitä ovat muun muassa seuraavat ominaisuudet:

- tarkkaan määritetyt pikseli tai kehyspuskuriobjektit
- käyttää uutta OpenGL ES Shading Language versiota 3.0
- useita rasterointikohteita samanaikaisesti
- standardoitu tekstuuri pakkausformaatti: ETC.

Ominaisuuksista ehkä merkittävimpänä voisi pitää GLSL ES -version uudistamista. Se toi listan uudistuksia varjostimissa käytettäviin koodiin. Esimerkiksi attribute- ja varying-tyyppiset muuttujat korvataan kokonaan in- ja out-tyyppisillä muuttujilla. Eli kun aikaisemmin kirjoitettiin varying vec2 vTexCoord, niin se pitää muuttaa muotoon out vec2 vTexCoord. Jotkin sisäänrakennetut muuttujat poistettiin myös. Esimerkiksi gl_FragColor-sisäänrakennettu muuttuja pitää korvata tekemällä oma muuttuja out vec4 fragColor. Sitä sitten käytetään main()-metodissa samanlailla kuin gl_FragColor-muuttujaa.

Koska GLSL ES 3.0 kielessä on näitä uusia ominaisuuksia, se tarkoittaa sitä, että jos sovellus kirjoitetaan GLSL ES -version 3.0 uusilla ominaisuuksilla, niin kohdelaite, joka

ei tue OpenGL ES 3.0:aa, eikä näin myöskään GLSL ES 3.0:aa, ei tule pyörimään kyseisessä laitteessa.

Toinen tärkeä uudistus on standardoitu tekstuuripakkausformaatti ETC. OpenGL ES:n ongelma oli pitkään ollut se, että sillä ei ollut standardoitua tekstuuripakkausformaattia. Se aiheutti sen, että toimittajien oli sovellettava heidän omia yhteensopimattomia tekstuuripakkausstandardeja. Suurien OpenGL ES GPU:en joukossa yleisimpiä tekstuuripakkausformaatteja ovat S3TC, PVRTC, ETC ja ATITC [20]. Koska OpenGL ES 2.0:ssa ei ole standardoitua tekstuuripakkausformaattia, kehittäjien on pakattava sovelluksien tekstuurit useaan kertaan eri laitteistolle, joka vie aikaa ja ennen kaikkea tilaa. Erityisesti Android-kehittäjille tämä on ongelma, koska Android-alusta tulee useita GPU: ita.

OpenGL ES -versio 3.1 julkaistiin 2014 maaliskuussa. Sekin on taaksepäin yhteensopiva versioiden 3.0 ja 2.0 kanssa. Se toi mukanaan joitain uusia ominaisuuksia, joita kehittäjät voivat lisäillä sovelluksiinsa:

- Laskentavarjostimet - Sovellukset voivat käyttää GPU: ta suorittamaan yleisiä laskentatehtäviä, jotka liittyvät tiukasti grafiikka-rasterointiin. Laskentavarjostimet on kirjoitettu GLSL ES -kielellä. [21.]
- Erilliset varjostinobjektit - Sovellukset voivat ajaa verteksi- ja fragmenttivarjostinvaiheet itsenäisesti, ja ne voivat ajaa verteksi- ja fragmenttivarjostinohjelmia ilman niille tarkoitettua linkitysvaihetta. [21.]
- Epäsuorat piirtokomennot – GPU:ta voidaan ohjata ottamaan piirtokomento GPU:n mapatusta muistista sen sijaan, että tieto kulkisi ajureiden kautta. Esimerkiksi tämä mahdollistaa GPU:lla suoritettavan laskentavarjostimen generoimaan tarvittavan piirtokomento informaation tuloksen näyttämiseen ilman CPU-synkronointia. [21.]

OpenGL ES -versio 3.2 taasen julkaistiin elokuussa 2015. Se toi mukanaan seuraavat ominaisuudet:

- Geometria ja tessellaatiovarjostimet käsittelevät tehokkaasti monimutkaisia kohtauksia GPU:ssa [11].
- Enemmän joustavuutta korkeamman tarkkuuden laskentaoperaatioissa [11].
- Kehittyneet tekstuurikohteet kuten tekstuuripuskurit ja ”multisample” 2D-taulukot [11].
- Uusia debuggauksen toimintoja kehittäjille [11].

# 4 Android Native Development Kit

Android-sovellukset ovat tyypillisesti kirjoitettu Javalla. Se ei ole kuitenkaan ainoa vaihtoehto kehittäjille, eikä esimerkiksi pelejä kehittäessä välttämättä se paraskaan. Native Development Kit (NDK) on joukko työkaluja, jotka antavat kehittäjille mahdollisuuden tehdä natiiveja Android-sovelluksia käyttäen C/C++-koodia. Se ei ole ehkä se paras vaihtoehto uusille Android-kehittäjille, mutta se on hyvä vaihtoehto sovelluksissa, jotka vaativat erittäin paljon prosessorilta tehoa erinäkösiin laskutoimituksiin. Tämä johtuu siitä, että lähdekoodi käännetään suoraan konekoodiksi prosessorille. Natiivin C/C++-koodin avulla pystytään myös ohittamaan Javan tuomat muistin hallinnan rajoitukset.

NDK tarjoaa myös joukon kirjastoja, joita kehittäjä voi käyttää esimerkiksi tapahtumankäsittelyyn. Kehittäjä pystyy määrittämään, mitä kirjastoja haluaa sovelluksessaan käyttää. Tämä on erityisen hyvä myös siitä syystä, että kehittäjä voi uudelleenkäyttää jotain aikaisempaa kirjastoa, mitä hän on käyttänyt jossain toisessa soveluksessa, tai sitten hän voi käyttää jonkun muun kehittäjän kirjastoa sovelluksessaan.

Kehittäjillä on muutama tapa rakentaa natiivi sovellus. Android Studio 2.2 -julkaisu toi mukanaan parannetun tuen ndk-build -nimiseen rakennustapaan, ja Cmake-nimisen rakennustavan natiivin koodin kääntämiseen. CMake on huomattavasti käytännöllisempi, joten siitä tulikin Android Studion oletustyökalu natiivin koodiin kääntämiseen. Android Studio kyllä tukee vielä ndk-buildia johtuen suuresta määrästä olemassaolevia projekteja, jotka on tehty nimenomaan ndk-buildilla. On kuitenkin suositeltavaa, että uusia sovelluksia tehdessä käytetään nimenomaan CMake-työkalua.

Uutta sovellusta tehdessä on vielä mahdollista tehdä se joko täysin natiivina eli pelkästään C/C++-koodin avulla, tai sitten käyttäen Java-koodia apuna. Tällöin esimerkiksi Javalla tehty aktiviteetti voi käyttää C/C++-koodilla tehtyjä natiiveja funktioita. Tämä on mahdollista Java Native Interfacesin avulla (JNI), joka mahdollistaa C/C++-koodin ja javakoodin keskenään käytävän keskustelun. Katsotaan seuraavaksi hieman tarkemmin, miten ndk-build ja CMake toimivat. Lisäksi katsotaan miten sovellus tehdään Javan ja natiivin koodin yhteistyöllä, sekä tutustutaan hieman miten Vulkan-sovellus tehdään täysin natiivilla koodilla.

## 4.1 Ndk-build -skripti

Ndk-build on komentoskripti, joka julkaistiin Android NDK r4: n yhteydessä. Sen tarkoitus on yksinkertaisesti käynnistää oikea NDK-rakennusskripti, jotta sovellus saadaan rakennettua. Ndk-build -skripti löytyy asennetusta NDK-hakemiston juuresta. Ndk-build skripti vaatii toimiakseen Android.mk-tiedoston, jossa kerrotaan projektin lähdekoodien sijainnit ja kirjastot, joita halutaan käyttää. Toinen tiedosto, joka on hyvä olla ndk-build -skriptiä käytettäessä on Application.mk, joka kertoo esimerkiksi, mitä moduuleja sovellus vaatii. Kun skripti ajetaan, niin sen oletushakemisto, mistä se etsii Android.mk-tiedostoa, on jni. Eli kun tekee Android.mk-tiedoston, on se hyvä laittaa kyseiseen hakemistoon. Toki se voi olla myös eri hakemisto, mutta se vaatii erillistä säätämistä asetuksista. Kun skripti on löytänyt Android.mk-tiedoston, se osaa automaattisesti etsiä Application.mk-tiedostoa samasta hakemistosta. Katsotaan esimerkkikoodia 18, jossa on määritetty yksinkertainen Android.mk-tiedosto.

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := gles2jni
LOCAL_SRC_FILES := gles2code.cpp
LOCAL_LDLIBS    := -llog -lGLESv2

include $(BUILD_SHARED_LIBRARY)
```

Esimerkkikoodi 18. Android.mk-tiedoston määrittely.

Android.mk-tiedoston pitää alkaa LOCAL_PATH-muuttujalla. Sen on tarkoitus kertoa, missä projektin lähdekoodit sijaitsee. Siinä kutsutaan makroa my-dir, joka kertoo, että ne sijaitsevat kyseisessä hakemistossa, eli siinä missä Android.mk-tiedostokin on. Seuraava rivi määrittää CLEAR_VARS-muuttujan. Se viittaa GNU Makefileen, joka resetoi kaikki LOCAL_XXX-muuttujat. Ainoa mitä se ei resetoi, on juuri äsken määritelty LOCAL_PATH-muuttuja. LOCAL_MODULE kertoo sen moduulin nimen, mikä halutaan rakentaa. LOCAL_SRC_FILES-kohdassa listataan kaikki lähdekoodit, mitä halutaan käyttää. Jos ne ovat samassa hakemistossa kuin Android.mk, riittää laittaa pelkkä tiedoston nimi. Jos sitä ei ole, niin tiedoston polku pitää määritellä myös. LOCAL_LDLIBS määrittää kaikki kirjastot, joita halutaan käyttää. Huomioitavaa on, että kirjastot vaativat –l etuliitteen. Eli esimerkiksi jos halutaan käyttää GLESv2 (OpenGL ES 2.0)-kirjastoa, määritetään se –lGKESv2. Viimeisenä include $(BUILD_SHARED_LIBRARY) kerää kaikki määritetyt LOCAL_* muuttujat, ja niiden tietojen perusteella päättää, mitä on rakentamassa ja miten sen tekee.

Application.mk-tiedostossa voi taas määritellä yksityiskohtaista tietoa siitä, mitä tullaan rakentamaan. Esimerkiksi voidaan määrittää, että halutaan rakentaa vain tietyt moduulit kaikista niistä, jotka Android.mk-tiedostossa on lueteltu. Jos halutaan käyttää staattista ajoa, voidaan se määrittää Application.mk-tiedostoon lauseella ’APP_STL := c++_static’. Tämä kuitenkin vaatii sen, että ollaan rakentamassa vain yhtä jaettua kirjastoa (vain yksi include(BUILD_SHARED_LIBRARY)-lause).

Ennen Android Studio 2.2 -julkaisua, piti ndk-buildia käyttää komentoriviltä. Skripti sijaitsi siis NDK-hakemiston juuressa, joten sitä pystyy sieltä käyttämään komennolla ’ndk-build <Projektin juuri>’. Tällöin se automaattisesti etsii Android.mk-tiedostoa jni-hakemistosta. Kun se sen löytää, se rakentaa kaikki siinä määritetyt kohdat ja tekee niistä .so-tiedoston (tai .a, jos kyseessä staattinen kirjasto). Kyseinen .so-tiedosto siis voidaan verrata Javan .jar-tiedostoon. Sitten rakennetaan Javan komponentit .dex-tiedostoon, ja lopulta pakataan ne kaikki (.so ja .dex) .apk-tiedostoon, jota voi käyttää androidissa. Kun Android Studio 2.2 julkaistiin, tuo se helpotusta kyseiseen prosessiin. Kun on itse tehnyt uuteen projektiin Android.mk-tiedoston johonkin kansioon, voi projektin juuritiedostoa (usein app) klikata hiiren oikealla, josta voi valita ’Link C++ Project with Gradle’. Tästä aukeaa ikkuna, josta voi valita rakennustavaksi joko Cmaken tai ndk-buildin. Lisäksi siinä määritellään se kansio, josta Android.mk-tiedosto löytyy. Sen jälkeen Android Studio tekee kaiken puolestasi (esim määrittää Gradleen ndkBuildPathin), jotta käyttäjä voi ajaa sovelluksen normaalisti Android Studiossa.

## 4.2 CMake

CMake-työkalu on yksinkertaisesti yksi skripti, joka on kirjoitettu CMakeList.txt-nimiseen tiedostoon. Skriptin tarkoitus on rakentaa kehittäjän C/C++-kirjastot. Kun kehittäjä tekee uutta Android Studio projektia, niin siinä voi valita vaihtoehdon ’include C++ support’, jolloin tiedosto tulee automaattisesti tehtyä projektiin. Jos sitä ei kuitenkaan ole, niin sen joutuu itse tekemään projektiin. Myös CMake pluginin joutuu asentamaan Android Studioon. Katsotaan esimerkkikoodissa 19 määritettyä CmakeList.txt-tiedostoa, jonka avulla voisi käyttää OpenGL ES 2.0:aa

```
# minimi version määritys.
cmake_minimum_required(VERSION 3.4.1)

# Laitetaan kääntäjän standardiksi c++11
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -Wall")
```

```
# luodaan kirjasto ja siihen liittyvät lähdekoodit
add_library(gl2triangle SHARED
            src/main/cpp/triangle.cpp
            src/main/cpp/triangle.h
            )

include_directories(src/main/cpp/)

#käytetään NDK API:n log-kirjastoa. Tallennetaan sen polku muuttujaan
find_library(log-lib log)

# lisätään omaan kirjastoon muita kirjastoja
target_link_libraries(gl2triangle
                      android
                      EGL
                      GLESv2
                      #liitetään tekeillä oleva kirjasto log kirjastoon
                      ${log-lib})
```

Esimerkkikoodi 19. CMakeList.txt-tiedoston määritys

Aluksi määritellään minimiversio CMakesta, mikä vaaditaan, jotta tekeillä oleva natiivi kirjasto saadaan rakennettua. Tämä takaa sen, että kaikki Cmaken uusimmat ominaisuudet ovat käytössä kirjastoa rakentaessa. Sen jälkeen flagien (kääntäjien asetuksien) avulla laitetaan kääntäjä käyttämään C++11-standardia. Add_library-komennolla luodaan se kirjasto, mitä ollaan tekemässä. Siihen aluksi laitetaan haluttu kirjaston nimi (esimerkkikoodissa 19 se on laitettu gl2triangle), ja valitaan, onko kirjasto shared tai static. Jos käytetään shared, niin kirjastot linkitetään dynaamisesti ja ladataan ajon aikana. Jos taasen valitaan static, niin kirjastot ovat objektitiedostojen arkistoja, joita voi käyttää, kun muita kohteita linkitetään. Tämän jälkeen laitetaan kaikki lähdekoodit, jotka kirjastoon halutaan. Jos lähdekoodit ovat samassa hakemistossa kuin missä CMakeList.txt sijaitsee, niin riittää kirjoittaa vain lähdekoodin tiedoston nimi. Jos lähdekoodi on eri hakemistossa, pitää tiedoston polku laittaa myös, kuten esimerkkikoodi 18:ssa on tehty. Jos käytetään header (.h)-tiedostoa, niin se pitää laittaa tähän myös. CMake myös käsittelee header-tiedostoja ulkopuolisina, joten jos niitä käyttää, niin tämän lisäksi pitää myös määrittää include_directories-kohdassa niiden polku. Kehittäjillä on mahdollisuus luoda niin monta kirjastoa kuin haluaa. Tarvitsee vain laittaa aina uusi add_library-lause CMakelist.txt-tiedostoon.

NDK tarjoaa myös joukon valmiita kirjastoja, joita kehittäjä pystyy käyttämään. Find_library-lausekkeella pystytään etsimään NDK:n tarjoamia kirjastoja, jotta niitä voi sitten käyttää omassa kirjastossaan. find_library-lausekkeessa sanotaan kirjaston nimi (toinen parametri, esimerkissä 19 etsitty log-kirjasto), ja tallennetaan se muuttujaan (log-lib). Tämän jälkeen pitää käyttää target_link_libraries-lauseketta, jossa määritellään, mitä kirjastoja CMaken halutaan liittävän kehittäjän omaan kirjastoon. Esimerkkikoodissa

18 on lausekkeeseen ensimmäiseksi määritetty se kirjaston nimi, johon kirjastoja halutaan liittää, eli add_library lausekkeessa määritetty gl2triangle. Sen jälkeen on käytetty android-kirjastoa, joka sisältää erinäisiä Androidin header-tiedostoja, esimerkiksi input eventit. Tämän jälkeen on liitetty EGL-kirjasto, joka tarjoaa natiivin alustan rajapinnan OpenGL ES -pintojen jakamiseen ja hallintaan. GLESv2 on itse OpenGL ES 2.0 -kirjasto. Lopuksi vielä liitetään etsitty log-kirjasto mukaan.

CMakelist.txt-tiedoston määrittelyn jälkeen pitää myös Androidin Gradle konfiguroida. Projektista löytyy module-tason build.gradle, jonka voi avata ja lisätä esimerkkikoodin 20 mukainen koodi.

```
externalNativeBuild {
    cmake {
        path 'src/main/cpp/CMakeLists.txt'
    }
}
```

Esimerkkikoodi 20. Gradlen konfigurointi.

Nyt ohjelmaa ajettaessa Gradle osaa suorittaa polkuun määritetyn CMake skriptin ja suorittaa siinä mainitut toimenpiteet.

## 4.3 Javan ja natiivin koodin yhteistyö

Kun CMake on konfiguroitu oikein, niin tarvitaan vielä Java ja natiivin koodi kommunikoimaan keskenään. Java-puolella voi OpenGL ES -sovellusta tehdessä tehdä normaalisti yhden aktiviteetin, GLSurfaceView-toteuttavan luokan sekä rasretointi-luokan. Jotta saadaan natiivi kirjasto käyttöön, niin on tehtävä myös esimerkiksi esimerkkikoodin 21 mukainen luokka.

```
public class GLES2JNILibrary {

     static {
          System.loadLibrary("gl2triangle");
     }

     public static native void init();
     public static native void resize(int width, int height);
     public static native void doDrawings();
}
```

Esimerkkikoodi 21. Luokka, joka antaa natiivin kirjaston käyttöön.

Luokassa käytetään System.loadLibrary()-metodia, jolla ladataan esimerkkikoodissa 19 CMakella määritetty gl2triangle-kirjasto käyttöön. Tämän jälkeen siinä on luotu natiiveja metodeja, joita Java voi käyttää. Sanalla native kerrotaan Javalle, että itse metodin toteutus on natiivissa koodissa, eli C/C++-koodin puolella, joka löytyy gl2triangle-kirjastosta.

Koska on tekeillä OpenGL ES -sovellus, voidaan määritettyjä kirjastoja käyttää siinä luokassa, joka toteuttaa GLSurfaceView.Renderer-rajapinnan esimerkkikoodin 22 mukaan.

```
public void onDrawFrame(GL10 unused) {
    GLES2JNILibrary.doDrawings();
}

public void onSurfaceChanged(GL10 unused, int width, int height) {
    GLES2JNILibrary.resize(width, height);
}

public void onSurfaceCreated(GL10 unused, EGLConfig config) {
    GLES2JNILibrary.init();
}
```

Esimerkkikoodi 22. Renderer-rajapinnan toteutus.

Tällöin kun sovellus käynnistetään, onSurfaceCreated()-metodissa tehdään natiivin koodin puolella init()-metodi, näytön koon muututtua tehdään resize()-metodi ja itse näytölle piirron tapahtuessa tehdään doDrawings()-metodi. Muistetaan yhä, että JNI on se rajapinta, jonka kautta Java- ja C++-komponentit keskustelevat keskenään.

Tarkastellaan metodien toteutusta natiivilla puolella, eli katsotaan, miten metodit pitää toteuttaa C/C++-puolella (esimerkkikoodin 18: n triangle.cpp tiedostossa) esimerkkikoodin 23 avulla.

```
extern "C" {

     JNIEXPORT void JNICALL Java_com_example_aleksi_ndktesti_GLES2JNILibrary_
init(JNIEnv * env, jobject obj);

     JNIEXPORT void JNICALL Java_com_example_aleksi_ndktesti_GLES2JNILibrary_
resize(JNIEnv* env, jobject obj, jint width, jint height);

     JNIEXPORT void JNICALL Java_com_example_aleksi_ndktesti_GLES2JNILibrary_
doDrawings(JNIEnv * env, jobject obj);

};

JNIEXPORT void JNICALL Java_com_android_gl2jni_GL2JNILib_init(JNIEnv * env,
jobject obj){
```

```
    create();
}

JNIEXPORT void JNICALL Java_com_android_gl2jni_GL2JNILib_resize(JNIEnv * env,
jobject obj,  jint width, jint height){

    setGraphics(width, height);
}

JNIEXPORT void JNICALL Java_com_example_aleksi_ndktesti_GLES2JNILibrary_
doDrawings (JNIEnv * env, jobject obj){

    renderFrame();
}
```

Esimerkkikoodi 23. Natiivien metodien määritys, jotta JNI löytää ne.

Huomataan esimerkkikoodista 23, että heti sen alussa on määritetty ’extern ”C”’. Kun JNI yrittää löytää Javan määrittämiä natiiveja funktioita automaattisesti, niin se ei C++-metodien nimiä osaa etsiä ylikuormitusvaaran takia. Ainoastaan määrittelemällä ’extern ”C”’ JNI osaa etsiä myös C++-funktioiden nimiä. Jos lause jää puuttumaan, niin JNI ei metodeja tule löytämään. ’Extern ”C”’-lausekkeessa määritetään ne natiivit funktiot, jotka halutaan JNI:n löytävän (samat kuin Javan puolella). Lausekkeessa aluksi on JNIEXPORT void JNICALL, joka vain määrittää, että natiivit funktiot ovat näkyvillä sekä että metodin muoto on void, eli se ei palauta mitään. Jos halutaan jotain Javaan palauttaa, niin sen tyyppi pitää laittaa voidin tilalle. Tämän jälkeen kirjoitetaan metodin sijainti. Jos katsotaan projektipuuta, niin GLES2JNILibrary-tiedosto sijaitsee Java-kansion alla, pakkauksessa nimeltä com.example.aleksi.ndktesti. Eli tuolla tiedolla pitää määrittää tiedoston sijainti laittamalla ensin kansion (java); erotinmerkkinä ’_’ määrittelee pakkauksen, sen jälkeen tiedoston nimen ja lopuksi itse metodin nimen. Parametreina pitää ainna olla JNIEnv * env (viittaus JNI-ympäristöön, jonka avulla voi käyttää kaikkia JNI-toimintoja) ja jobject obj (sama kuin javassa ’this’-määrite).

Kun lausekkeet on määritetty löydettäväksi, pitää niiden toteutus vielä toteuttaa. Samalla tavalla jokaiseen lausekkeeseen vain laitetaan toteutus sisälle. Esimerkiksi doDrawings()-metodi toteuttaa renderFrame()-metodin kyseisessä .cpp-tiedostossa.

Katsotaan vielä renderFrame()-metodia, joka piirtää kolmion kuvaruudulle esimerkkikoodissa 24.

```
void renderFrame() {

     glClearColor(0.5F, 0.5F, 0.5F, 0.5F);
     glClear(GL_COLOR_BUFFER_BIT);
```

```
    //Kaytetaan luotua ohjelmaa
    glUseProgram(myProgram);

    mMVPMatrixHandle = (GLuint) glGetUniformLocation(myProgram, "uMVP-
Matrix");
    mPositionHandle = (GLuint) glGetAttribLocation(myProgram, "v_Position");

    //Laitetaan identiteetti matriisille, ja laitetaan se tiettyyn kohtaan
    mModelMatrix->identity();
    mModelMatrix->translate(0.0f, -1.0f, 0.0f);

    glVertexAttribPointer(
           (GLuint) mPositionHandle,
           3,
           GL_FLOAT,
           GL_FALSE,
           0,
           verticesData
    );
    glEnableVertexAttribArray((GLuint) mPositionHandle);

    // Lisätään mMVPMatriisiin tehty näkymä matriisi ja modelmatriisi
    mMVPMatrix->multiply(*mViewMatrix, *mModelMatrix);

    // Lisätään mMVPMatriisiin vielä määritetty projektiomatriisi
    mMVPMatrix->multiply(*mProjectionMatrix, *mMVPMatrix);

    //Sovelletaan tehtyä matriisia
    glUniformMatrix4fv(mMVPMatrixHandle, 1, GL_FALSE, mMVPMatrix->mData);

    //piirretään kolmio ruudulle
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Esimerkkikoodi 24. Kolmion piirto ruudulle käyttäen C++-natiivia koodia.

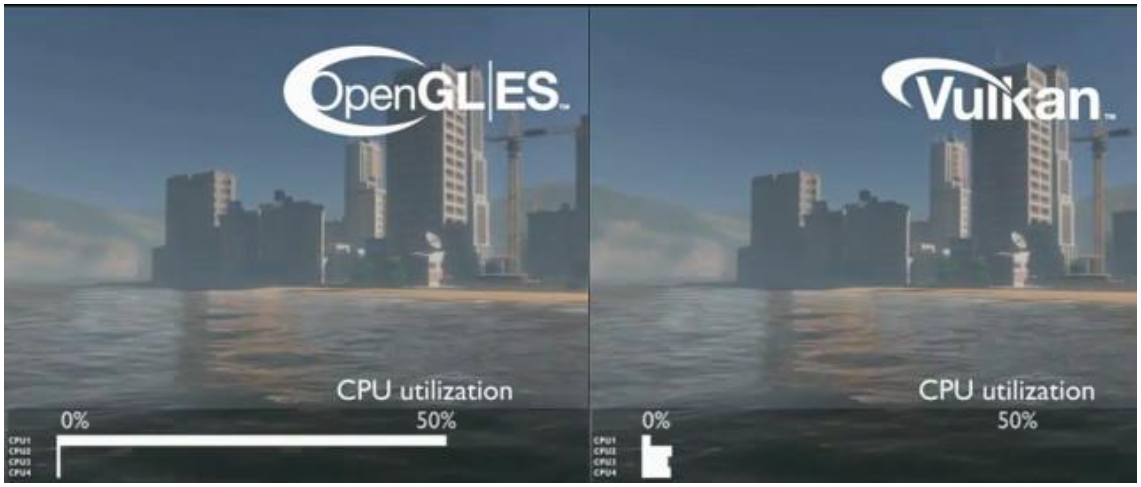
Kolmion piirto menee C++-puolella käytännössä aika samalla tavalla kuin se menisi Javan puolella. Oleellisena erona voidaan pitää, että toisin kuin Javassa, buffereita ei tarvitse tehdä ollenkaan. Kolmiota piirtäessä ensin määritetään ja ladataan varjostimet (C++11-standardi toi mukanaan automuuttujan, joka automaattisesti muuttaa syötetyn tekstin oikeaan tyyppiin, joka sitten korvaa automuuttujan), määritetään verteksit, tehdään ohjelmaobjekti ja piirretään kolmio ruudulle esimerkkikoodin 24 mukaan. Huomioitavaa on se, että koska GLESv2-kirjasto oli lisätty omaan tehtyyn kirjastoon CMakella, niin nyt sen metodeja pystytään käyttämään, esimerkiksi glDrawArrays()-metodia.

## 4.4 Vulkan

Vulkan on Khronos Groupin vuoden 2016 helmikuussa julkaisema API. Sitä voi käyttää korkealaatuisten ja hyvän suorituskyvyn omaavien 3D-sovelluksien, kuten pelien

tekemiseen. Vulkan API:n nimen piti alun perin olla ’OpenGL Next’ tai ’Next generation OpenGL’, mutta lopulta päädyttiin nimeen Vulkan. Sen olisikin tarkoitus jonakin päivänä korvata OpenGL kokonaan. Vuonna 2013 AMD kehitti API:n nimeltä Mantle, joka toimi myös rajapintana 3D-sovelluksien tekoon. Vuonna 2015 AMD kuitenkin päätti itse lopettaa sen kehittämisen, ja päätti samalla lahjoittaa Mantlen Khronos Groupille, jotta se voisi rakentaa Vulkanin Mantle:n pohjalta. [22] Koska Vulkan julkaistiin vasta vuonna 2016, tarkoittaa se sitä, että se vaatii API-tason 24 toimiakseen. Eli jos haluaa Android-puhelimessa käyttää Vulkania, niin se tarvii vähintään Android-versio 7: n eli Nougatin. Vaaditaan myös vähintään NDK-versio r12. Vulkan toimii myös useissa alustoissa kuten Windowsissa, Linuxissa ja Androidissa.

Verrattuna OpenGL: ään Vulkanissa on monia asioita pyritty tekemään paremmin. Siinä on esimerkiksi enemmän suoraa kontrollia grafiikkaprosessoriin, eli koodi on ns. ’lähempänä metallia’, sekä siinä on pienempi prosessorin kulutus. Vulkan pystyy myös jakamaan työnsä paremmin useiden CPU-ytimien kesken. Mikä on myös käytännöllisempää Vulkanissa, on se, että se ei ole sidottu yhteen tilaan niin kuin OpenGL, joka on käytännössä vain yksi globaalinen tilakone. Vulkan on sen sijaan objektipohjainen, jossa ei ole ollenkaan globaalia tilaa. Katsotaan kuvaa 16, josta huomataan, miten Vulkan osaa paremmin jakaa prosessorin työtä useiden CPU-ytimien kesken. Tällöin se tuottaa vähemmän kuormitusta yksittäiselle prosessorille.




Kuva 16. OpenGL ES:n ja Vulkanin prosessorin tehon jakautumisen vertailu [23].

### 4.4.1 Vulkan-sovelluksen toimintaperiaate

Jotta voisi saada Vulkan-sovelluksen näkymään ruudulla, pitää käyttää pelkästään natiivia koodia käyttäen NativeActivityä. NativeActivity on Androidin sisäänrakennettu Java-luokka, jonka avulla kehittäjä voi toteuttaa sovellukset puhtaasti natiivin koodin avulla. Tämä on Vulkan sovelluksen kannalta hyvinkin hyödyllistä, sillä jotta Vulkanin swapchain saadaan rakennettua, tarvitaan ANativeWindow-kahvan , jonka NativeActivity tarjoaa kehittäjän käyttöön. Swapchain on taas se tapa, jolla Vulkan-sovellukset saadaan näkymään ruudulle.

Jotta saadaan NativeActivity käyttöön, pitää AndroidManifest.xml-tiedostoa hieman säätää. Esimerkkikoodissa 25 on tehty AndroidManifestiin tarvittavat toimenpiteet.

```
<application android:label="@string/app_name"
android:icon="@drawable/ic_launcher" android:hasCode="false">
     <activity android:name="android.app.NativeActivity"
          android:label="@string/app_name">
          <meta-data android:name="android.app.lib_name" />
          android:value="native"
          <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
     </activity>
</application>
```

Esimerkkikoodi 25. Manifestin laittaminen Vulkanille sopivaan kuntoon.

Tärkein osa manifestissa on laittaa android.app.NativeActivity sovelluksen käynnistymisaktiviteetiksi, mihin samalla laitetaan android.app.lib_name meta-datan paikalle. Kyseinen meta-data kertoo sen jaetun kirjaston nimen, mikä halutaan ladata. Todetaan myös, että kehittäjän ei tarvitse käyttää ollenkaan Javaa koodissaan, joten voidaan laittaa android:hasCode arvoon false.

Sovelluksessa pitää myös käyttää staattista kirjastoa, joka toimii rajapintana NativeActivitylle. Sen nimi on android_native_app_glue.c (ja .h), ja se löytyy asennetusta NDK:n hakemistosta. Itse sovelluksessa pitää toteuttaa metodi nimeltä android_main(), jota kutsutaan, kun sovellus käynnistetään ensimmäisen kerran. Se saa parametrina pointerin android_app-nimiseen structiin, joka sisältää viittauksia toisiin tärkeisiin elementteihin, kuten ANativeWindow-kahvaan. Se sisältää myös ALooper-instanssin, joka kuuntelee kahta asiaa. Aktiviteetin elinkaaren toimintoja, kuten pause ja resume, sekä se kuuntelee aktiviteetin syöttötapahtumia.Kun ANativeWindow-kahva on luotu

metodissa, voi sen laittaa sovellukseen näkymäksi, jolloin sovelluksella on näkymä, mihin voi piirtää asioita.

Tämän jälkeen voidaan tehdä Vulkan-ilmentymä VkInstance, joka toimii ensimmäisenä sisääntulona Vulkaniin sekä kuvastaa ”loader”:ia. Kyseisestä ilmentymästä voidaan sitten tiedustella käytössä olevat grafiikkaprosessorit systeemissä. Kun sopiva grafiikkaprosessori on valittu, voidaan siitä tehdä laitemuuttuja (VkDevice). Tämä Vulkan-laitemuuttuja toimii lähtökohtana työn lähettämiselle grafiikkaprosessorille. Kun laitemuuttuja on tehty, voidaan tehdä swapchain. Sinne voidaan tallentaa erilaisia kuvia tai muotoja yms. Jotta ne saadaan näkymään näytölle, tehdään ensin näyton pinta VKSurfaceKHR. Kun pinta on tehty, otetaan swapchainistä haluttu kuva ja näytetään se näytön pinnalla.

### 4.4.2 Varjostimien toteutustapa

Vulkan-sovelluksen pitää käsitellä varjostimia eritavalla kuin OpenGL ES:n. OpenGL ES:ssä tehdään varjostin tekstijoukkona, jotka muodostavat GLSL-varjostin-ohjelman lähdekoodin. Vulkan API sen sijaan edellyttää, että kehittäjä antaa varjostimen SPIR-V-moduuliin sisääntulopisteen muodossa. SPIR-V on korkean tason kieli, joka esiintyy binäärimuodossa. Yksi binäärimuodon etu on se, että grafiikkaprosessorien toimittajien kirjoittamat kääntäjät, joiden tarkoitus on kääntää varjostimen koodi natiiviksi koodiksi, on huomattavasti vähemmän monimutkaista. On huomattu- että ihmisen luettavalla syntaksilla, kuten GLSL:llä, jotkut grafiikkaprosessorien toimittajat olivat melko joustavia standardin tulkitsemisessa. Tämä tarkoitti sitä- että jos sattui kirjoittamaan ei niin triviaaleja varjostimia jollekin kyseiselle grafiikkaprosessorille, niin on vaarana- että toisten toimittajien ajurit hylkäävät koodin syntaksivirheiden vuoksi, tai jopa se, että varjostin toimisi eri tavalla johtuen käännösvirheistä. SPIR-V:n suoraviivaisella binäärikoodiformaatilla tältä ongelmalta pitäisi välttyä.

Itse varjostimen kirjoittaminen Vulkan-sovellukseen tapahtuu melkein samalla tavalla kuin OpenGL ES:ssä. Katsotaan esimerkkikoodia 26, jossa yksinkertainen verteksivarjostin Vulkanille on kirjoitettu.

```
#version 400
#extension GL_ARB_separate_shader_objects : enable
layout (location = 0) out vec4 uFragColor;
void main() {
```

```
   uFragColor = vec4(0.1, 0.2,  0.3, 1.0);
}
```

Esimerkkikoodi 26. Vulkanille kirjoitettu verteksivarjostin.

Huomataan, että kehittäjän ei tarvitse kirjoittaa binäärikoodia käsin, vaan koodin voi aluksi kirjottaa GLSL-kielellä. Jotta sen tunnistaa Vulkanille sopivaksi, rivi GL_ARB_separate_shader_objects pitää lisätä. Khronos on julkaissut grafiikkaprosessorin toimittajista riippumattoman kääntäjän nimeltä glslang, joka kokoaa ja kääntää GLSL:n SPIR-V:lle. Tämä kääntäjä on suunniteltu varmistamaan, että varjostion-koodi on täysin vaatimusten mukainen ja tuottaa yhden SPIR-V-binäärin, jota voi ohjelmassa käyttää. Jotta Android Studio Vulkan-ohjelmaa kääntäessä löytää varjostimet, pitää niiden sijaita hakemistossa app/src/main/shaders/. Lisäksi Android Studio tunnistaa varjostimen tyypin tiedostopäätteen mukaan. Verteksivarjostin pitää olla .vert-loppuinen, ja fragmenttivarjostin .frag-loppuinen. Tällöin glslang kääntää tiedostot SPIR-V-muotoon, jota sovellus sitten käyttää.

### 4.4.3 Validointikerrokset

Useimmat grafiikka-API:t eivät suorita virhetarkistuksia, koska ne voivat johtaa sovelluksen suorituksen hidastumiseen. Vulkan tarjoaa virheidentarkistuksen tavalla, jonka avulla kehittäjä voi käyttää virheidentarkistusominaisuutta sovelluksen kehityksen aikana, mutta lopullisessa valmiissa sovelluksessa helposti ottaa sen pois päältä, täten maksimoidaan sovelluksen suorituksen nopeus. Tämä toteutetaan tekemällä ja laittamalla päälle validointikerrokset. Nämä validointikerrokset hakevat Vulkan-sisääntulopisteitä eri virheenkorjaus- ja validointitarkoituksiin.

Jokainen validointikerros voi sisältää määritelmiä yhdelle tai useammalle sisääntulopisteelle. Täten se voi puuttua jokaiseen sisääntulopisteeseen, johon validointikerroksella on määritelmiä. Kun validointikerrokseen ei ole määritelty sisääntulopistettä, sovellus siirtää sisääntulopisteen seuraavalla kerrokselle. Jos joku sisääntulopiste ei ole määriteltynä missään validointikerroksessa, se on sovelluksessa ilman validointia.

NDK r12 ja myöhemmät versiot sisältävät valmiiksi rakennettuja validointikerroksien binäärejä. Kun sovelluksessa tehdään Vulkan-ilmentymä, voi sovelluksessa määritellä, että ilmentymä eli ”loader” hakee kyseiset validointikerrokset, ja lataa ne käyttöön.

Android Studiossa voi projektin Gradlea konfiguroimalla saada validointikerrokset käyttöön sovellukseen. Esimerkkikoodissa 27 on projektin Gradleen vaadittavat operaatiot.

```
sourceSets{
     main{
          jniLibs{
               srcDir ”${oma-ndk-
hakemisto}/sources/third_party/vulkan/src/build-android/jniLibs”
          }
     }
}
```

Esimerkkikoodi 27. Projektin Gradlen säätäminen validointikerroksien hakuun.

Katsotaan ja tutkitaan hieman esimerkkikoodia 28, jossa laitetaan muutama validointikerros käyttöön sovelluksessa.

```
//otetaan kerroksien lukumäärä ylös
uint32_t instance_layer_present_count = 0;
vkEnumerateInstanceLayerProperties(&instance_layer_present_count, nullptr);

// Varataan tilaa äsken lasketun countin verran
VkLayerProperties* layer_props = (VkLayerProperties*)malloc(in-
stance_layer_present_count * sizeof(VkLayerProperties));

// otetaan jokaisen olemassa olevan kerroksen ominaisuudet talteen
vkEnumerateInstanceLayerProperties(&instance_layer_present_count,
layer_props));

// Listataan itse halutut kerrokset, jotka halutaan käyttöön
const char *instance_layers[] = {
    "VK_LAYER_GOOGLE_threading",
    "VK_LAYER_LUNARG_parameter_validation",
    "VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_core_validation",
    "VK_LAYER_GOOGLE_unique_objects"
};

//Tarkistetaan, että jokainen äske määritelty haluttu kerros on olemassa
uint32_t instance_layer_request_count =
    sizeof(instance_layers) / sizeof(instance_layers[0]);
for (uint32_t i = 0; i < instance_layer_request_count; i++) {
     bool found = false;

     //Katsotaan for-silmukassa että jokainen kerros tosiaan löytyy
     //katsotaan että haluttu nimi löytyy alussa etsityn layer_props taulukon
     //kerroksen ominaisuudesta layerName
     for (uint32_t j = 0; j < instance_layer_present_count; j++) {
          if (strcmp(instance_layers[i], layer_props[j].layerName) == 0) {
               found = true;
          }
     }
     //Jos ei löydy, laitetaan error tulemaan
     if (!found) {
          error();
     }
}
```

```
// Laitetaan halutut kerrokset ilmentymän luonnin yhteyteen
VkInstanceCreateInfo instance_info = {};
instance_info.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
instance_info.enabledLayerCount = instance_layer_request_count;
instance_info.ppEnabledLayerNames = instance_layers;
```

Esimerkkikoodi 28. Validointikerroksien käyttöönotto Vulkanissa

Esimerkkikoodin 28 alussa huomataan, että vkEnumerateInstanceLayerProperties()-metodi on avainasemassa. Siinä katsotaan, montako kerrosta on tarjolla ja otetaan lukumäärä ylös (muistetaan että ne löytyi NDK-hakemistosta). Sitten varataan malloc()-metodilla muistia kyseisen lukumäärän verran, ja sitten layer_props-muuttujaan otetaan jokaisesta kerroksesta ominaisuudet talteen. Tämän jälkeen kirjoitetaan taulukko, missä ovat kaikki halutut validointikerrokset. Pitää kuitenkin tarkistaa, että jokainen haluttu kerros on löytynyt, joten for-silmukassa verrataan jokaista haluttua kerrosta, layer_props-taulukon (siinä oli kaikki löydetyt kerrokset) layerName-arvoon, jossa on kerroksen nimi. Jos kerrosta ei löydy, tuotetaan virhe. Lopuksi Vulkan ilmentymän yhteydessä laitetaan siihen enableLayerCount siihen arvoon, montako kerrosta on, ja sitten vielä ppEnableLayerNames kohtaan laitetaan se taulukko, jossa kerroksien nimet on.

Kun kerrokset on määritelty, pitää vielä laittaa Debug Report -laajennus VK_EXT_debug_report-käyttöönD, jotta sovellus voi hallita kerroksien käyttäytymistä virheen sattuessa. Ensin pitää varmistaa, että alusta tukee laajennusta, jonka jälkeen se otetaan käyttöön ja laitetaan tuottamaan mahdollisia virheraportteja sinne, minne kehittäjä haluaa.

# 5 3D-pelisovelluksen toteuttaminen

Tässä luvussa käsitellään 3D-pelisovelluksen tekoa OpenGL ES 3.0:n avulla. Tutustutaan 3D-tekniikoihin työtä varten tehdyn pelisovelluksen avulla. Tarkoitus oli pitää itse peli hyvinkin yksinkertaisena (2-vuotias poikani pääsi läpi), mutta kuitenkin sellaisena, että siitä saa hyvän idean siitä, miten 3D-sovelluksen voi tehdä OpenGL ES 3.0:n avulla. Projektissa käytetään tuttuun tapaan Android Studio-ohjelmaa, jolla voi Android-sovelluksia tehdä.

## 5.1 Pelin idea

Pelin idea on hyvin yksinkertainen. Aloitusruudulle tulee 6 eri 3D-muotoparia (kuva 17).


Kuva 17. Pelin aloitusruutu.

Kun käyttäjä painaa jotain muotoa näytöltä, alkaa se pyörimään. Pelin tarkoituksena on painaa kahta samaa muotoa, jolloin ne molemmat pyörivät. Tällöin muodot lähtevät pyöriessään lähestymään käyttäjää, eli suurenevat niin paljon, että ne lopulta häviävät ruudulta. Jos käyttäjä on valinnut kaksi samaa muotoa, ja myös jonkin toisen muodon, niin silloin ne eivät lähde lähestymään käyttäjää, eli häviämään ruudulta. Käyttäjällä pitää olla valittuna ainoistaan yksi muotopari, jotta sen saa näytöltä pois. Kuvassa 18 näkyy pelitilanne, jossa pyramidi on saatu pois näytöltä, ja kuutio on lähestymässä käyttäjää, eli häviämässä.

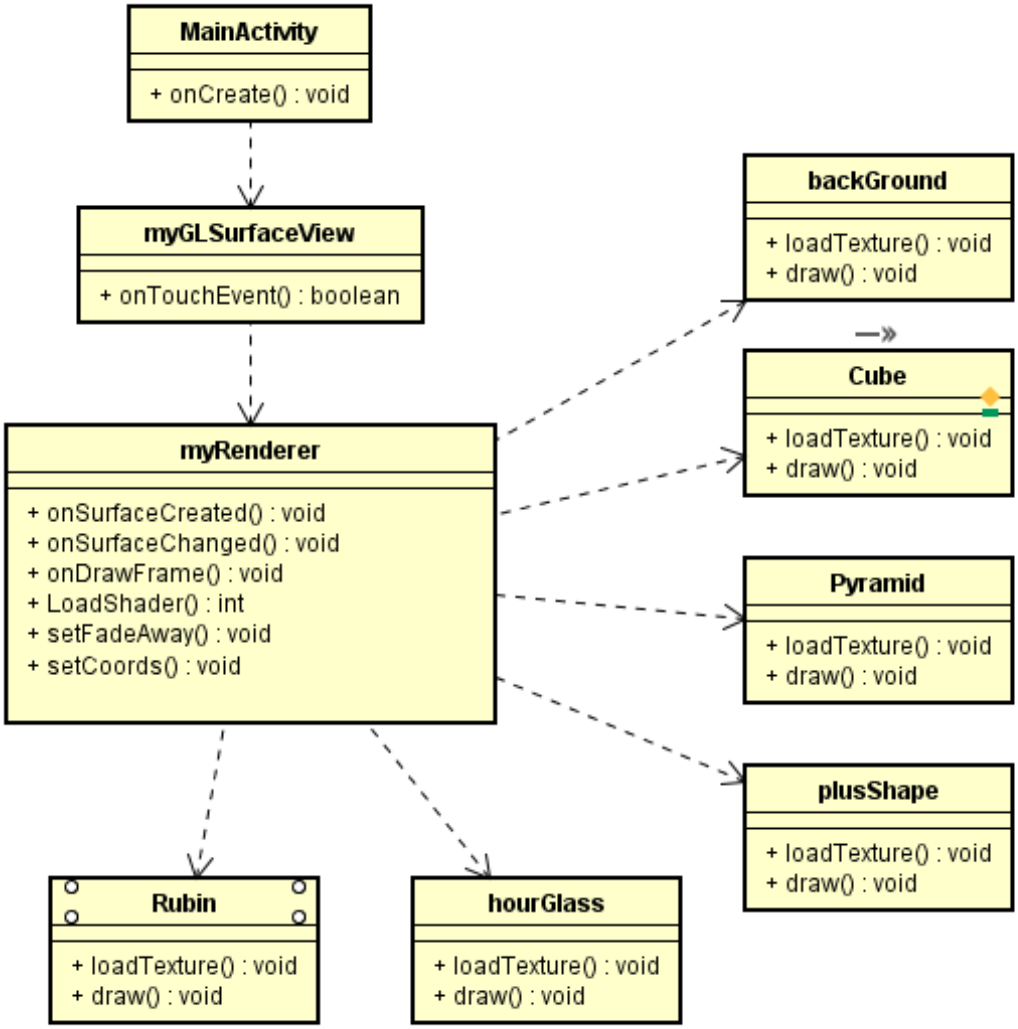

Kuva 18. Kuutio poistumassa ruudulta.

Kun käyttäjä on saanut kaikki muodot pois ruudulta, tulee ruudulle onnitteluviesti (samalla tekstityylillä kuin aloitusruudussa), ja mahdollisuus aloittaa peli alusta.

## 5.2 Rakenne

Peli koostuu yhdestä aktiviteetistä, johon laitetaan myGLSurfaceView-luokan näkymä. Kyseiseen luokkaan liitetään myRenderer-luokan toteutus, joka sitten rasteroi näkymää erilaisilla piirtokomennoilla. MyGLSurfaceView-luokkaan myös laitetaan onTouchEvent()-metodi, jossa käsitellään kaikki käyttäjän tekemät toimenpiteet ruudulla. Koska peli on tehty OpenGL ES 3.0:lla, myRenderer-luokan pitää ensinnäkin implementoida GLSurfaceView.Renderer-luokkaa, sekä siinä pitää olla toteutettuna onSurfaceCreated(), onSurfaceChanged() ja onDrawFrame()-metodit.

Jokainen eri muoto on toteutettu omana luokkanaan. Ne alustetaan onSurfaceCreated()-metodissa, ja lopulta onDrawFrame()-metodissa jokainen muoto piirretään niiden omalla draw()-metodilla. Kuvassa 19 nähdään pelin luokkakaavio.




Kuva 19. Pelin luokkakaavio.

## 5.3 3D-muotojen piirtäminen

Kun lähdetään piirtämään 3D-muotoja, tarvitaan x- ja y-koordinaattien lisäksi säätää myös z-koordinaattia, joka antaa muodolle syvyyden. Jos haluaa piirtää jotain hyvinkin monimutkaisia muotoja, kehittäjän kannattaisi käyttää siihen soveltuvia työkaluja kuten esimerkiksi Blender-ohjelmaa. Sillä voisi piirtää muodon valmiista komponenteista ja muodostaa siitä .obj-tiedoston, jota sitten voisi Android Studiossa käyttää. Tässä työssä tosin tehdään kaikki muodot pelkästään Android Studiolla. Tarkastellaan pyramidin muotoisen muodon luokkaa hieman tarkemmin. Esimerkkikoodissa 29 on koordinaatit, joista pyramidimuoto muodostuu.

```
//pyramidin koko muuttujana, jotta sitä on helppo muuttaa halutessaan
private float size = 0.2f;

private float[] verticesCoords = new float[]{
        // Yläpiste
        0.0f, size, 0.0f, // top

        // Alaneliön 4 verteksiä.
        -size, -size, -size,
        -size, -size, size,
        size, -size, size,
        size, -size, -size,

        //Alaneliön 4'stä verteksistä 2 uudestaan,
        //jotta niihin saadaan eri tekstuurikoordinaatit
        //alaneliön tekstuuria varten
        size, -size, size,
        size, -size, -size,

};
private short[] indices = {
        //Yläpuolen kolmiot
        1, 0, 2, 2, 0, 3, 3, 0, 4, 4, 0, 1,
        //alapuolen neliö
        1, 6, 5, 5, 2, 1,

};
private float textureCoords[] = {

        0.5f , 0,
        0, 1,
        1, 1,
        0, 1,
        1, 1,

        //alaneliön tekstuurit
        1, 0,
        0, 0,
};
```

Esimerkkikoodi 29. Pyramidin koordinaatit

Muistetaan kuvassa 3 oleva androidin koordinaattisysteemi, jota koordinaateissa sovelletaan. Kaikilla vertekseillä pitää olla x- , y- ja z-koordinaatit. Pyramidi muodostuu 5 eri verteksistä eli pisteestä (pyramidin huippu piste, ja alaneliön 4 pistettä), joiden välille viivan vetämällä saa pyramiidin piirrettyä. Indeksien avulla (indices) määritetään se piirtojärjestys, joilla muoto saadaan piirrettyä. Eli käytännössä siinä piirretään 4 eri kolmiota pyramidin sivuihin, sekä myös kaksi kolmiota, joista neliönmuotoinen pohja muodostuu. Lisäksi esimerkkikoodi 29:ssä on tehty tekstuurikoordinaateille oma taulukko, jotta tekstuurit saadaan piirrettyä. Huomioitavaa on se, että jotta alaneliön tekstuuri saataisiin hyvin piirrettyä, pitää kaksi verteksiä tehdä toiseen kertaan, jotta niille saadaan oikeat tekstuurikoordinaatit (1,0 ja 0,0).

Muotojen koordinaatteja miettiessä kannattaa piirtää esimerkiksi paperilla x- ,y- ja z-koordinaatisto, jotta eri verteksien koordinaatit voi helpommin hahmottaa ja saada oikein.

Indeksien avulla onneksi ei tarvitse kuin jokainen muodon piste kerran miettiä, joista voi sitten piirtää muodon. Verteksien määrä kasvaisi huomattavasti ilman indeksejä.

Kun koordinaatit on pyramidissa saatu kuntoon, tarvitaan verteksi- ja fragmenttivarjostimet. Esimerkkikoodi 30:ssä nähdään pyramidin tarvittavat varjostimet.

```
//verteksivarjostimen koodi
private String vShaderStr =
        "#version 300 es           \n"
                + "uniform mat4 uMVPMatrix;     \n"
                + "in vec4 vPosition;           \n"
                + "in vec2 TexCoordIn;          \n"
                + "out vec2 TexCoordOut;    \n"
                + "void main()                  \n"
                + "{                            \n"
                + "   gl_Position = uMVPMatrix * vPosition;  \n"
                + "  TexCoordOut = TexCoordIn;"
                + "}                            \n";
//fragmentti varjostimen koodi
private String fShaderStr =
        "#version 300 es                          \n"
                + "precision mediump float;                    \n"
                + "uniform sampler2D TexCoordIn; \n"
                + "out vec4 fragColor;                     \n"
                + "in vec2 TexCoordOut;                      \n"
                + "void main()                                      \n"
                + "{                                                \n"
                + " fragColor = texture(TexCoordIn, TexCoordOut); \n"
                + "}                                                \n";
```

Esimerkkikoodi 30. Varjostimien koodit käyttäen OpenGL ES –versiota 3.0.

Koska käytetään OpenGL ES -versiota 3.0, käytetään siinä myös OpenGL Shading Language -versiota 3.0. Tällöin OpenGL ES -version 2.0 attribute- ja varying-muuttujat on korvattu in- ja out-muuttujilla. Verteksivarjostin saa verteksien koordinaatit vPosition-muuttujaan, ja tekstuurikoordinaatit TexCoordIn-muuttujaan. TexCoordOut-muuttuja taas välittää tekstuurikoordinaattitiedot fragmenttivarjostimeen, joka käyttää sitä sampler2D-tekstuurityypin tekemiseen. OpenGL ES 2.0:n gl_fragColor on myös korvattu fragColor-nimisellä out-muuttujalla. Varjostimien alussa täytyy myös laittaa #version 300 es -määritelmä. Se enabloi GLSL-versio 3.0 käytön. Samalla se antaa virheilmoituksen ”version ’300’ is not supported” laitteille, jotka eivät tue OpenGL ES -versiota 3.0.

Varjostimien ja koordinaattien luonnin jälkeen tehdään Pyramidi-luokan konstruktorissa bufferit vertekseille, tekstuureille ja indekseille. Tämän jälkeen jo käsiteltyyn tyyliin ladataan verteksi- ja fragmenttivarjostimet, tehdään OpenGL-ohjelma objekti GLES30.glCreateProgram()-metodilla, liitetään varjostimet siihen ja lopuksi linkitetään ohjelma GLES30.glLinkProgram(programObject)-metodilla.

Kun muotojen luokat alustetaan onSurfaceCreated()-metodissa, ladataan siinä myös luokille tekstuurit loadTexture()-metodilla. Se käyttää Android Studio-projektin drawable-kansiossa olevia kuvia, jotka se esimerkkikoodi 8:n tapaan lataa käyttövalmiiksi (GL10 korvattu tietenkin GLES30:lla).

OnSurfaceChanged()-metodissa laitetaan näyttö mukautumaan puhelimen kääntöä varten esimerkkikoodin 31 mukaan.

```
public void onSurfaceChanged(GL10 unused, int width, int height) {

    // Näkökentän luonti
    GLES30.glViewport(0, 0, width, height);
    float aspectRatio = (float) width / height;

    Matrix.perspectiveM(projectionMatrix, 0, 53.00f, aspectRatio, 1, 10);
}
```

Esimerkkikoodi 31. Projektiomatriisin luonti

Esimerkkikoodin 31 metodissa laitetaan näkökenttä alkamaan vasemmasta yläkulmasta näytön leveyden ja korkeuden verran. Samalla luodaan perspektiivinen projektiomatriisi, joka tallennetaan 16-paikkaiseen mProjectionMatrix-taulukkoon.

Itse piirto tapahtuu onDrawFrame()-metodissa. Esimerkkikoodissa 32 tapahtuu pyramidin matriisien määrittely.

```
//otetaan systeemin ylhäälläolo aika
long time = SystemClock.uptimeMillis() % 10000L;
float angleInDegrees = (360.0f / 10000.0f) * ((int) time);

GLES30.glClear(GLES30.GL_COLOR_BUFFER_BIT | GLES30.GL_DEPTH_BUFFER_BIT);
//Pakollinen päälle laitto, jotta syvyys saadaan näkymään
GLES30.glEnable(GLES30.GL_DEPTH_TEST);

//Laitetaan kameran positio. Eli se kohta mistä kamera katsoo näkökenttään
Matrix.setLookAtM(viewMatrix, 0, 0, 0, 0.5f, 0f, 0f, -5.0f, 0f, 1.0f, 0.0f);
//Laitetaan matriisin identiteetti.
Matrix.setIdentityM(shapeMatrix, 0);
//Laitetaan positio. Tällä laitetaan piirrettävän pyramiidin position näytöllä
Matrix.translateM(shapeMatrix, 0, 0.0f, 1.0f, -3.5f+fadeAway[1]);
//Laitetaan rotaatio, eli käännetään kappaletta 320 astetta kerran.
Matrix.rotateM(shapeMatrix, 0, 320f, 0.4f, 1.0f, 0.6f);
if(isRotating[1]){
     //tehdään jatkuvaa kiertoa akselin ympäri
     Matrix.rotateM(shapeMatrix, 0, angleInDegrees, 1.0f, 1.0f, 1.0f);
}

if(conditionMet[1]){
     setFadeAway(1, 9);
}
//lisätään kaikki tehdyt matriisit yhteen ainoaan matriisiin
Matrix.multiplyMM(mMVPMatrix, 0, viewMatrix, 0, shapeMatrix, 0);
```

```
Matrix.multiplyMM(mMVPMatrix, 0, projectionMatrix, 0, mMVPMatrix, 0);
pyramid.draw(mMVPMatrix);
```

Esimerkkikoodi 32. Pyramidin matriisien määrittely

Ensinnäkin esimerkkikoodissa 32 on enabloitu GL_DEPTH_TEST. Tämä laittaa syvyyden näkymisen päälle, jotta mikä tahansa 3D-muoto voisi näkyä oikein ruudulla. Laitetaan kameran sijainti näkökenttään ja luodaan näkymämatriisi setLookAtM()-metodilla. Tehdään matriisin identiteetti setIdentityM()-metodilla, eli tavallaan nollataan matriisi ja laitetaan tulevat transformaatiot siihen matriisiin. Matrix.translateM()-metodi on tärkeä, sillä siinä laitetaan pyramidin sijainti ruudulla. Siinä määritellään pyramiidin x-, y- ja z-koordinaatit. Huomioitavaa on tässä z-koordinaatti, joka on laitettu -3.5f. Eli se näkyy -3.5 arvon verran kaukaisuudessa. Sovelluksessa on tehty myös avaruudennäköinen taustakuva, joka on vain yksi tekstuuri, joka on laitettu näkymään koko näytön leveydeksi. Mutta se näkyy taustalla muiden muotojen takana sen takia, koska sen z-koordinaatin arvo on -5.0f. Tällöin se näkyy kauempana, ja muut muodot lähempänä, eli tavallaan taustakuvan edessä. Pyramiidin z-koordinaatin arvoon on myös liitetty fadeAway-arvo (taulukossa on 12 alkiota, joista jokainen kuvaa yhtä muotoa. Pyramidi on taulukon alkio 1). Kun conditionMet[1] ehto täyttyy(molemmat pyramiidit pyörii), laittaa setFadeAway()-metodi kasvattamaan muotojen 1 ja 9 (kaksi pyramiidia) fadeAway-arvoja tietyn väliajoin hieman. Tällöin muoto tulee lähemmäksi käyttäjää ruudulla, kunnes se katoaa pois.

Jokaiselle muodolle tehdään myös Matrix.rotateM()-metodi, joka pyörittää tietyn astemäärän verran kappaletta tietyssä suunnassa. Kuten esimerkkikoodissa 32 huomataan, pyramidiin on tehty yksi 320 asteen kierto tietyssä suunnassa. Tämän takia aloitusruudussa kuvassa 16 pyramidi ja kaikki muut kuviot näkyvät hieman kääntyneinä, eikä normaalisti pystysuorassa. Jos ehto isRotating[1] täyttyy, eli käyttäjä on painanut pyramidin muotoa, alkaa pyramidi jatkuvasti pyörimään. Tämä on toteutetty angleInDegreed-muuttujan avulla. SystemClock.uptimeMillis()-metodi palauttaa millisekuntteina sen arvon, mitä systeemi on ollut päällä sitten viime bootin. Tämän avulla lasketaan angleInDegreed-muuttujaan arvot 0 - 360f. Sen on tarkoitus kymmenen sekunnin välein käydä läpi arvot 0 – 360. Tällöin kappaleet saavat yhden täydellisen rotaation akselinsa ympäri kerran kymmenessä sekunnissa.

Lopuksi kaikki tehdyt matriisit lisätään yhteen mMVPMatriisiin (Model-View-Projection) Matrix.multiplyMM()-metodilla. Tämän jälkeen tehty matriisi annetaan parametrina

pyramiidi muodon draw()-metodille, joka tekee piirron ruudulle. Katsotaan esimerkkikoodia 33, jossa piirto tapahtuu.

```
public void draw(float[] mvpMatrix) {

      GLES30.glUseProgram(mProgramObject);

      int mMVPMatrixHandle = GLES30.glGetUniformLocation(mProgramObject, "uMVP-
Matrix");

      int mPositionHandle = GLES30.glGetAttribLocation(mProgramObject, "vPosi-
tion");

      int vsTextureCoord = GLES30.glGetAttribLocation(mProgramObject, "TexCoor-
dIn");

      // sovelletaan transformaatio ja näkymä matriisia
      GLES30.glUniformMatrix4fv(mMVPMatrixHandle, 1, false, mvpMatrix, 0);

      GLES30.glEnableVertexAttribArray(vsTextureCoord);
      GLES30.glEnableVertexAttribArray(mPositionHandle);

      GLES30.glVertexAttribPointer(mPositionHandle, COORDS_PER_VERTEX,
            GLES30.GL_FLOAT, false,
            vertexStride, vertexBuffer);

      GLES30.glVertexAttribPointer(vsTextureCoord, COORDS_PER_TEXTURE,
            GLES30.GL_FLOAT, false,
            textureStride, textureBuffer);

      GLES30.glActiveTexture(GLES30.GL_TEXTURE0);
      GLES30.glBindTexture(GLES30.GL_TEXTURE_2D, textures[0]);

      //Piirretään pyramiidi ruudulle
      GLES30.glDrawElements(GLES30.GL_TRIANGLES, indices.length, GLES30.GL_UN-
SIGNED_SHORT, drawListBuffer);

}
```

Esimerkkikoodi 33. Pyramidin piirto matriisien avulla.

Draw()-metodin alussa laitetaan OpenGL ES käyttämään aikaisemmin pyramidiluokan konstruktorissa luotua ohjelmaobjektia. Otetaan kyseisestä ohjelmasta käyttöön matriisi, sekä viittaukset vektorien ja tekstuurien positioista. GLES30.glUniformMatrix4fv()-metodilla otetaan parametrina saadusta, aikaisemmin luodusta matriisista projektio ja näkökenttätransformaatiot, ja lisätään ne ohjelmasta käyttöön otettuun matriisiin. GLES30.glVertexAttribPointer()-metodilla kerrotaan OpenGL ES -säikeelle tiedot vertekseistä ja tekstuureista. Se saa parametrina aikaisemmin otetun vektori-viittauksen, COORDS_PER_VERTEX-muuttujan, joka on siis 3, koordinatin tyypin joka on vertekseissa float-tyyppinen, verteksien stride-arvon, joka siis viittaa siihen muistin määrään, mitä yhden verteksin piirtoon tarvitaan. Koska yhdessä verteksissä on 3 koordinaattia, ja jokainen niistä vie 4 tavua tilaa, niin täytyy vertexStride-arvon olla 3*4

eli 12. textureStride on samalla tavoin 2*4 eli kahdeksan. Lopuksi metodi saa tietenkin viittauksen luotuun verteksibufferiin.

Tämän jälkeen aktivoidaan ja vielä sidotaan luotu tekstuuri, joka sijaitsee siis texture-taulukon ensimmäisessä alkiossa. Tämä sama tekstuuri piirretään pyramidin jokaiselle 4 sivulle, sekä myös pohjaneliöön. Itse piirto tapahtuu GLES30.glDrawElements()-metodilla. Siinä kerrotaan, että ollaan kolmioita piirtämässä, annetaan indeksien määrä, kerrotaan, että ne ovat short-tyyppisiä sekä annetaan viittaus luotuun indeksien bufferiin.

## 5.4 Tapahtumankäsittely

Kuten jo aikaisemmin mainitsin, myGLSurfaceView-luokassa täytyy toteuttaa onTouchEvent()-metodi. Siinä voidaan käsittellä kaikki käyttäjän tekemät tapahtumat näytöllä. Esimerkkikoodissa 34 on pelisovelluksessa käyttämäni tapahtumankäsittelijä.

```
@Override
public boolean onTouchEvent(MotionEvent e) {
    float x = getWidth() / 2;
    float y = getHeight() / 2;

    switch (e.getAction()) {

        case MotionEvent.ACTION_DOWN:

            float realX = e.getX() / x - 1;
            float realY = e.getY() / y - 1;

            myRender.setCoords(realX, realY);

    }

    return true;
}
```

Esimerkkikoodi 34. Tapahtumankäsittelijän toteutus.

Kun käyttäjä painaa sormella näytön tiettyä kohtaa, tulee painalluksesta tietty positio. Tarkoituksenani oli muuttaa painalluksen positio täsmäämään androidin koordinaattisysteemiä, jossa ruudun vasen laita saa arvon -1, ja oikea positiivisen ykkösen. Sama juttu tehdään korkeuden suhteen. Tämä on toteutettu ensinnäkin ottamalla talteen käyttäjän laitteen näytön korkeus ja leveys. Sanotaan, että näytön leveys on 1080. Jos käyttäjä painaa x-akselin kohdasta 810 (3/4 leveydeltä), niin silloin halutaan saada realX arvoon 0,5. Tämä olisi myös tällöin androidin koordinaattisysteemissä 3/4 leveys. Tämä saavutetaan jakamalla näytön leveys kahtia,

jolloin saadaan x arvoon 540. E.getX()-metodi palauttaa käyttäjän painalluksesta tulleen position, eli sen 810. Tämä jaetaan saadulla 540 arvolla, josta vähennetään 1 (jotta päästään negatiivisiin arvoihin), jolloin saadan realX:n oikeaksi arvoksi 0,5. Tällöin saadaan sama 0,5 arvo riippumatta siitä, minkälainen käyttäjän näytön leveys on.

MotionEvent.ACTION_DOWN tulee voimaan, kun käyttäjä painaa sormella ruutua. Siinä lasketaan painalluksen koordinaatit ja myRenderer-luokkaan annetaan painalluksen koordinaatit.

# 6 Yhteenveto

Android on kovalla vauhdilla noussut ylivoimaisesti suosituimmaksi mobiilien käyttöjärjestelmäksi viime vuosina. Kovan suosion ansiosta onkin tärkeää ymmärtää ne tekniikat, millä näyttävän näköisiä graafisia pelejä voidaan ohjelmoida. Tämän työn tarkoituksena olikin perehtyä kyseisiin eri tekniikoihin, joilla saadaan graafisia sovelluksia aikaiseksi. Tavoitteena oli saada hyvä käsitys OpenGL ES -kirjastosta ja sen kehityksestä, sekä tutkia, miten uusimmilla OpenGL ES -versioilla saadaan yksinkertainen 3D-peli ohjelmoitua. Lisäksi työssä tutkittiin miten graafisia sovelluksia saadaan aikaiseksi NDK:n työkaluilla.

Työssä käytiin OpenGL ES -kirjaston eri versiot ja niillä sovelluksen toteuttaminen läpi. Huomattiin, että jos on uudempi kehittäjä, jolla ei ole hirveästi kokemusta OpenGL ES -kirjastosta, haluaa tehdä hyvän graafisen sovelluksen, niin kannattanee käyttää OpenGL ES -versiota 1.0/1.1. Todettiin, että se on kaikista versioista se yksinkertaisin toteuttaa. Myöhemmät versiot 2.0:sta ylöspäin toivat uusia haasteita kehittäjille. Niissä kehittäjä pystyy melkein täysin vaikuttamaan graafisen sovelluksen jokaiseen osa-alueeseen varjostimien avulla. Tämä tuo mukanaan jonkin verran lisää ohjelmoimista, joten niiden toteuttaminen on hankalampaa. Työssä tehtiin uusimmalla OpenGL ES -versiolla oma 3D-peli, josta katsottiin ja todettiin käytännössä, mitä näyttävän 3D-pelin tekeminen vaatii. Kyseisessä pelissä todettiin myös, miten 2D-sovelluksesta siirrytään 3D-sovelluksen pariin.

Työn toisessa osiossa perehdyttiin myös Android Native Development Kittiin. Tutkittiin, mitä eri työkaluja se sisältää ja miten niillä toteutetaan sovelluksia. Todettiin että NDK:n avulla kehittäjä pystyy tekemään natiiveja sovelluksia C/C++-kielen avulla. Huomattiin

myös, että JNI:n avulla kehittäjä pystyy tekemään Javan ja natiivin koodin yhteistyöllä hienoja sovelluksia. Osion lopussa perehdyttiin vielä Vulkan-nimiseen ohjelmointirajapintaan. Todettiin, että jos kehittäjä haluaa maksimoida sovelluksen suorituskyvyn, kannattaa sen ehdottomasti käyttää Vulkania OpenGL ES:n sijaan.

Loppupäätelmänä voisi sanoa, että graafisen sovelluksen tekeminen Android Studiolla vaatii hieman perehtymistä, mutta kun esimerkiksi OpenGL ES -kirjaston toiminnan sisäistää, niin voi saada suhteellisen pienellä vaivalla näyttäviä graafisia sovelluksia aikaiseksi. Jonkin verran monimutkaisempia ovat NDK:n avulla tehdyt sovellukset, koska joutuu menemään Javan mukavuusalueelta C/C++-koodin pariin. Varsinkin Vulkanissa kehittäjä joutuu jokaisen pienenkin yksityiskohdan itse ohjelmoimaan C/C++-koodilla. Tosin jotta sovelluksen suorituskyky ja näyttävyys saadaan maksimoitua, kannattaa kehittäjän ehdottomasti tehdä uudet graafiset sovellukset Vulkan-ohjelmointirajapinnalla. Vulkan onkin äskettäin julkaistu ohjelmointirajapinta, niin on mielenkiintoista nähdä, miten paljon Vulkanin suosio kasvaa tulevaisuudessa verrattuna OpenGL ES:ään.

## Lähteet


1 Beavis, Gareth. 2008. A complete history of Android. Verkkoaineisto. Techradar. <http://www.techradar.com/news/phone-and-communications/mobile-phones/a-complete-history-of-android-470327>. 23.09.2008. Luettu 26.1.2018.

2 Android (operating system). 2018. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Android_(operating_system)>. Luettu 26.1.2018.

3 Open Handset Alliance. 2018. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Open_Handset_Alliance>. Luettu 26.1.2018.

4 Cheng, Jacqui. 2011. Android tops everyone in 2010 market share; 2011 may be different. Arstechnica. <https://arstechnica.com/gadgets/2011/01/android-beats-nokia-apple-rim-in-2010-but-firm-warns-about-2011/>. 31.01.2011. Luettu 29.1.2018.

5 Smartphone OS Market Share, 2017 Q1. 2017. Verkkoaineisto. IDC. <https://www.idc.com/promo/smartphone-market-share/os>. Luettu 31.01.2018.

6 Dalvik (software). 2018. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Dalvik_(software)>. Luettu 03.02.2018.

7 Android Runtime. 2018. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Android_Runtime>. Luettu 03.02.2018.

8 Android version history. 2018. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Android_version_history>. Luettu 05.02.2018.

9 Most popular Android versions in December 2017. 2017. Verkkoaineisto. Fossbytes. <https://fossbytes.com/most-popular-android-versions-always-updated/>. Luettu 6.2.2018.

10 Khronos Group. 2018. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/Khronos_Group>. Luettu 12.2.2018.

11 OpenGL ES. 2018. Verkkoaineisto. Wikipedia. < https://en.wikipedia.org/wiki/OpenGL_ES>. Luettu 12.2.2018.

12 Opengles Pipeline. 2018. Verkkoaineisto. < https://sites.google.com/site/face2manoj/opengl/pipeline>. Luettu 15.2.2018.

13 OpenGL ES. 2018 Verkkoaineisto. Android Developer. < https://developer.android.com/guide/topics/graphics/opengl.html>. Luettu 15.2.2018.

14 Zechner, Mario. 2011. Beginning Android Games. New York: Apress, Inc.

15 The Quad textured. 2018. Verkkoaineisto. Wiki (lwjgl). < http://wiki.lwjgl.org/wiki/The_Quad_textured.html>. Luettu 3.3.2018

16 OpenGL Programming Guide. 2018. Verkkoaineisto. glProgramming. < http://www.glprogramming.com/red/chapter03.html#name3>. Luettu 5.3.2018

17 Zen Bound 2 Universal Review. 2010. Verkkoaineisto. SlidetoPlay. < http://www.slidetoplay.com/zen-bound-2-review/>. Luettu 6.3.2018

18 Tutorial 3 : Matrices. 2018. Verkkoaineisto. Opengl-tutorial. < http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/>. Luettu 8.3.2018

19 Android Lesson Four: Introducing basic Texturing. 2018. Verkkoaineisto. Learn OpenGL ES. http://www.learnopengles.com/android-lesson-four-introducing-basic-texturing/>. Luettu 19.3.2018

20 What's New in OpenGL ES 3.0. 2012. Verkkoaineisto. AnandTech. < https://www.anandtech.com/show/6134/khronos-announces-opengl-es-30-opengl-43-astc-texture-compression-clu/2>. Luettu 20.3.2018

21 Khronos Releases OpenGL ES 3.1 Specification. 2014. Verkkoaineisto. Khronos. <https://www.khronos.org/news/press/khronos-releases-opengl-es-3.1-specification>. Luettu 20.3.2018

22 Vulkan (API). 2018. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Vulkan_(API)>. Luettu 12.4.2018.

23 Embedded system news. 2017. Verkkoaineisto. Cnxsoft. <https://www.cnx-soft-ware.com/2016/10/20/this-video-shows-vulkan-apis-higher-power-efficiency-com-pared-to-opengl-es-api/>. Luettu 12.4.2018.